UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Interval Joins for Big Data

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Eldon Preston Carman, Jr.

September 2020

The Dissertation of Eldon Preston Carman, Jr. is approved:

Committee Chairperson

University of California, Riverside

## Acknowledgments

I am grateful to my advisor, Professor Vassilis Tsotras, without whose support, I would not have been able to complete this journey. Thank you for your continued encouragement and guidance through this learning experience. I especially would like to thank my dissertation committee members, Dr. Ahmed Eldawy, Dr. Vagelis Hristidis, Dr. Eamonn Keogh, and Dr. Michael Carey, for reviewing my dissertation. I also like to thank Dr. Rajiv Gupta for his support in the beginning of this dissertation. I would like to thank everyone in AsterixDB's team for their support. Especially Professor Michael Carey for his guidance in understanding the AsterixDB stack and managing the research in an active software product. Thank you to my lab partners, Ildar Absalyamov and Steven Jacobs, for all the lively discussions on our regular drives out to Irvine. Apache VXQuery had a strong base created by Vinayak Borkar and Till Westmann which made Chapter 7 possible and their continued feedback and guidance on how to develop Apache VXQuery. I would also like to acknowledge the other members of the AsterixDB team: Young-Seok Kim, Taewoo Kim, Jianfeng Jia, Pouria Pirzadeh, Abdullah Alamoudi, Murtadha Hubail, Yingyi Bu, Ali Alsuliman, Raman Grover, Sattam Alsubaiee, Keren-Audrey Ouaknine, and Dmitry Lychagin. Thanks to the rest of the UCR Database lab: Moloud Shahbazi, Jarod Wen, Shiwen Cheng, Mohiuddin Qader, Christina Pavlopoulou, James Feng, and Elena Strzheletska. Thank you to Susan Gardner and James Foster for their editorial feedback.

The text of this dissertation, in part, is a reprint of the material as it appears in IEEE Big Data 2015. The co-author Vassilis Tsotras listed in that publication directed and supervised the research which forms the basis for this dissertation.

This research was partially supported by NSF grants: IIS-0910859, IIS-1447826 and CNS-1305253. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

To my wife and family for all their support.

In the words of my daughter Cassia, “I love you so much!”

ABSTRACT OF THE DISSERTATION

Interval Joins for Big Data

by

Eldon Preston Carman, Jr.

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2020
Dr. Vassilis J. Tsotras, Chairperson

The main part of this dissertation considers how to scale interval join queries. To provide scalable query processing for such joins, we adapted five recently published overlapping interval join algorithms and modified them to work in a shared-nothing big data management system (AsterixDB) under a memory budget. We developed a cost model for each algorithm to predict when an algorithm will spill to disk (run out of memory). Our experimental evaluation shows that the cost models are accurate and can be used to pick the most efficient algorithm for the given input data. The adapted interval join algorithms are shown to scale for large datasets using both synthetic and real datasets. Finally, we further adapt these algorithms to support several new types of interval joins, specifically overlap and contains, as defined by Allen's interval algebra. We detail how to abstract the memory management from these algorithms.

As a by-product we also implemented a scalable parallel processor, namely Apache VXQuery, that extends a stack consisting of Hyracks, a parallel execution engine, and Algebricks, a language-agnostic compiler toolbox. VXQuery provides an implementation of the XQuery specifics (data model, data-model dependent functions and optimizations, and a parser). We describe the architecture of Apache VXQuery, its integration with Hyracks and Algebricks, and the XQuery optimization rules applied to the query plan to improve path expression efficiency and to enable query parallelism. An experimental evaluation using a real 500GB dataset with various XML selection, aggregation, and join queries shows that Apache VXQuery performs well both in terms of scale-up and speed-up.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Today's applications create many types of data. In this dissertation, we consider two types of data: data with intervals and data stored in XML. For the first data type, consider jobs running on a supercomputer, songs played on a streaming site, and network files transferred as a few examples where the activity includes a start and end time, thus creating an interval that describes the duration of the activity. The existence of interval data leads to interesting queries. One of the most processing-intensive queries is the 'interval-join', where tuples from two relations are joined together if their respective intervals satisfy some query-specified condition. There are various join conditions between two intervals (overlaps, covers, covered_by, etc.), as defined by Allen's relationships [2]. The second type of data comes from the widespread acceptance of XML as a standard for document management and data exchange, which has enabled the creation of large repositories of XML data. An interesting problem is to efficiently query such large data collections while taking advantage of parallelism.

In this dissertation we consider scenarios where data is very large (as, for example, in log-related applications that keep track of what happened in an application logs). As a result, solutions to interval joins or XML data need to scale to the data size. The dissertation starts by looking at interval join and then wraps up by considering XML data. While there are many recent works presenting various interval-join algorithms, they are limited to running on a single (possibly multi-core) machine, and often all in main memory. Here we consider how these algorithms can be extended to work in a standard big data environment, where data can reside on many different processing nodes with bounded memory working, under a shared-nothing framework. In particular, we implement our interval join algorithms on AsterixDB, an open-source, shared-nothing distributed environment. In Chapter 2 we review the shared-nothing architecture of AsterixDB and its basic join query structure.

One basic interval join condition is for the tuple intervals to overlap; we call this query the 'overlapping interval join' [29] and it is the focus in Chapter 3. Given two relations whose tuples include intervals, an overlapping interval join finds all pairs of tuples from the two relations whose intervals share at least one time instant. Since a given tuple's interval may overlap with many intervals in the other relation, the join results tend to be large, often larger than the input relations.

The state-of-the-art algorithms for interval overlapping joins can be divided in two categories; ones that further partition data to perform the interval join and those that do not use partitioning. The non-partitioning interval join algorithms – sort-merge (SM) (as described by [38]), time-sweep (TS) (a modified algorithm from [43]) and forward scan (FS) [16] – and the partitioning interval join algorithms–overlap interval partition join (OIP)

[24] and disjoint interval partitioning (DIP) [17]–are reviewed in Chapter 3. Chapter 3 also looks at each algorithm's memory usage and how to run the algorithm when memory is insufficient for in-memory operations. Experiments were performed to show how different interval properties affect these algorithms and consider both varied data input and query output characteristics. Since each algorithm uses its memory budget in different ways, we created cost models to enable these algorithms to be integrated into a cost-based query optimizer. Chapter 4 describes the cost models for predicting the processing associated with Memory, CPU, and IO. We include a model for determining the join size estimation, which predicts how many tuples will be created from an interval join.

Recent publications on interval joins have focused on the overlapping join condition. There are, however, many types of interval joins, as defined by Allen's thirteen interval algebra relationships [2]. These thirteen relationships define all possible ways that one interval can relate to another interval and thus allow for more descriptive interval queries. In Chapter 6, we describe how to extend the five state-of-the-art interval join algorithms to process four of Allen's interval algebra relationships, namely: *covers, covered-by, overlaps,* and *overlapped-by.* We focus on these four relationships since their predicates involve both the start and end points of the two intervals; relationships like *starts*, *finishes*, or *meets* involve one interval end point and are thus easier to process – using, for example, traditional indexes like B-trees or hashing. The five state-of-the-art interval join algorithms have been updated to process four of Allen's interval algebra relationships. Using Sort Merge Interval Join as a test case, we show experiments that explore how these Allen interval joins are impacted through speed up, scale up and handle limited memory.

Chapter 7 turns from intervals to semistructured data and in particular to how to build a scalable processor for querying XML data, using the same interval frameworks that support AsterixDB. To efficiently query such large XML data collections, a scalable implementation of XQuery (the standard XML query language) is needed that can take advantage of parallelism. The result is the Apache VXQuery processor, which builds upon two other open-source Asterix frameworks: Hyracks, a parallel execution engine, and Algebricks, a language agnostic compiler toolbox. Apache VXQuery provides an implementation of the XQuery specifics (data model, data-model dependent functions and optimizations, and a parser) and is currently available as open source at the Apache Software Foundation [13]. We describe the architecture of Apache VXQuery, its integration with Hyracks and Algebricks, and the XQuery optimization rules applied to query plans to improve path expression efficiency and to enable query parallelism. We have performed an experimental evaluation using a large (500GB) real dataset (a NOAA weather dataset from [14]) and various XML selection, aggregation, and join queries that show the efficiency of our parallel XQuery processor, both in terms of speed up and scale up.

# Chapter 2

# Background: The AsterixDB Software Stack

In this chapter, we introduce the AsterixDB software stack and explain how a traditional (i.e., non-interval) join query is currently optimized and executed. This chapter will serve as a background for Chapters 3 and 6, where we describe how AsterixDB was extended to support interval joins. AsterixDB's software stack can be represented in three layers, as shown in Figure 2.1. After parsing a supplied SQL++ query statement, the top layer, AsterixDB, builds an Algebricks logical plan. The Algebricks logical plan is then optimized and translated into an Algebricks physical plan that maps directly to a Hyracks job. A brief explanation of each layer in the stack follows in the next subsections. Figure 2.1 also shows how other systems, such as the Apache VXQuery processor [33], can use the layers of the Algebricks and Hyracks infrastructure. Chapter 7 will present details of building the Apache VXQuery processor using Algebricks and Hyracks.

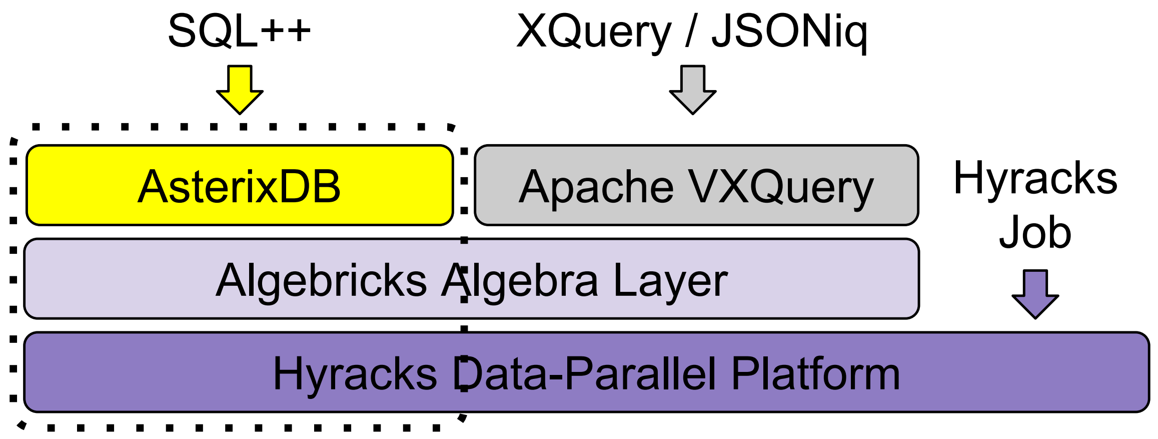


Figure 2.1: The layers of the Asterix architecture.

AsterixDB is an open-source, shared-nothing distributed platform for big data management. A shared-nothing architecture consists of many nodes where each node has its own processes, memory, and disks. AsterixDB runs on a cluster of commodity hardware. A cluster controller manages the cluster and handles incoming queries' requests. Figure 2.2 shows the components of the AsterixDB cluster and node controllers which work together to orchestrate the query execution. In the following sections we will describe these pieces, starting from the bottom of the stack.

## 2.1 Hyracks

Hyracks is a data-parallel execution platform that builds upon mature parallel database techniques and modern big data trends [12, 30]. This generic platform offers a framework to run dataflows in parallel on a shared-nothing cluster. The system was designed to be independent of any particular data model. Hyracks processes data in partitions of contiguous bytes, moving data in fixed-sized frames that contain physical records. It

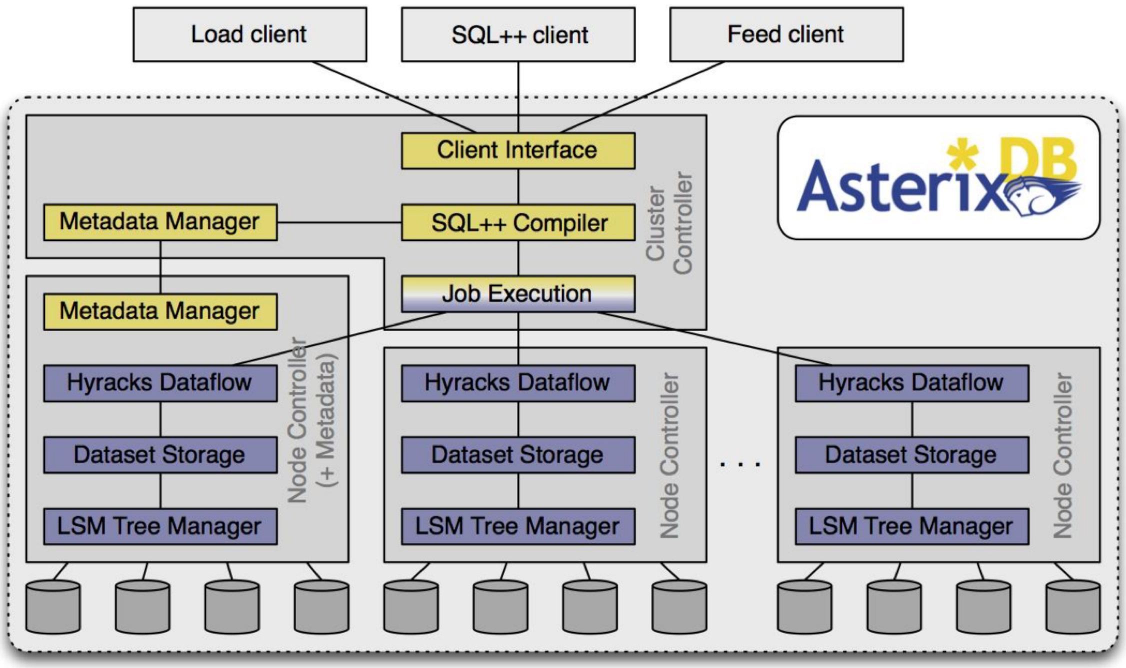


Figure 2.2: The components of the AsterixDB system architecture.

also defines interfaces that allow users of the platform to specify the data-type details for comparing, hashing, serializing and de-serializing data. Hyracks provides built-in base data types to support storing data on partitions or when building higher level data types (first row of Table 2.1).

A Hyracks job is defined by a dataflow directed acyclic graph (DAG) with operators (nodes) and connectors (edges). During execution, the operators allow the computation

| | |
|---|---|
| Hyracks Base Types | boolean, byte, byte array, short, integer, long, double, float, UTF8 string, void |
| AsterixDB Types | Interval, List, Record |

Table 2.1: AsterixDB builds on the Hyracks Base types to create more advanced data types.

to consume an input partition and produce an output partition while the connectors re-distribute data among partitions. The dataflow among Hyracks operators is push-based: each source (producer) operator pushes the output frames to a target (consumer) operator. The extensible runtime platform provides a number of operators and connectors for use in forming Hyracks jobs. While each operator's operation is defined by Hyracks, the operator relies on data-model specific functionality provided by the client of (next level above) the platform.

## 2.2 Algebricks

Algebricks [13, 14] is a parallel framework providing an abstract algebra for parallel query translation and optimization. This language-agnostic toolbox complements the lower-level extensible Hyracks platform. Implementations of data-intensive programming languages can extend Hyracks' model-agnostic algebraic layer to create parallel query processors on top of the Hyracks platform. A language developer is free to define the language and data model when using the Hyracks platform and the Algebricks toolkit. Algebricks features a rule-based optimizer and data-model-neutral operators that allow for language specific customization. Figure 2.3 shows each of the components that Algebricks provides and the components contributed by the language implementation.

A system that uses Algebricks for its query processing provides its own parser and translator to translate a query to a query plan that uses Algebricks' logical operators as an intermediate representation. The Algebricks rule-based optimizer then transforms the query plan over three stages. The first is a Logical-to-Logical plan optimizer that creates alternate

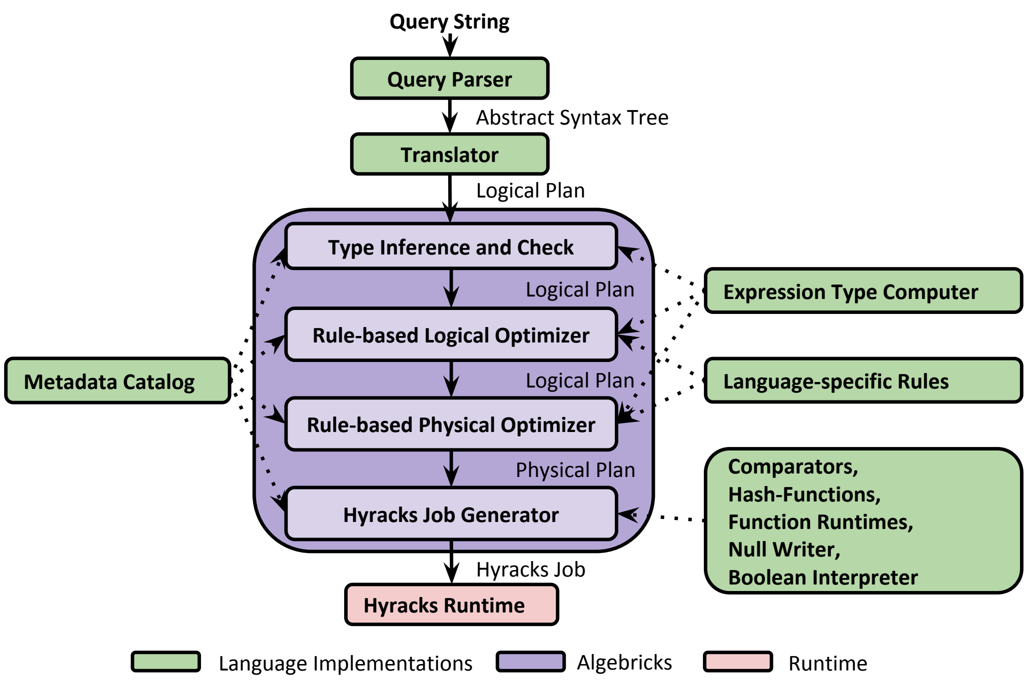


Figure 2.3: The components of the Algebricks architecture.

logical plans. Once the logical plan is finalized, the Logical-to-Physical plan optimizer converts the logical operators into a physical plan. Then, the physical optimizer considers the operator characteristics, partition properties, and data locality to choose the optimal physical implementation for the plan. Algebricks provides generic language-independent rewrite rules for each stage and allows for the addition of other rules. Finally, a Hyracks job is generated and submitted for execution on a Hyracks cluster.

Algebricks' intermediate logical algebra uses logical operators that map onto Hyracks' physical operators. A logical operator's properties are considered when determining the best physical operator. For example, a join query that has an equijoin predicate allows a hash-based join instead of the default nested-loop join. The Algebricks logical operators exchange data in the form of logical tuples, each of which is a set of fields. The following Algebricks logical operators are used in a basic AsterixDB query plan:

The DATASCAN operator reads from a data source and returns one tuple for each item in the data source.

The DISTRIBUTE-RESULT operator collects the final query results on each participating data node. Once the job is completed, the controller will request each local result and transfer it back to the user to create a complete result.

The EMPTY-TUPLE-SOURCE operator contributes the first tuple without any fields. Algebricks uses this operator to start all DAG dataflow paths.

The JOIN operator matches and combines tuples from two streams of input tuples.

The ORDER operator sorts tuples in the local partition.

The Algebricks operators are each parameterized with custom expressions. The expressions map directly to functions provided by the higher level (which can be built-in functions or custom functions). These operators are linked by connectors that are responsible for transporting data from one operator to the next. The following are common connectors:

The 1:1 EXCHANGE connector reads from an operator and sends the data to another operator.

The 1:1 PARTITION EXCHANGE connector reads from an operator and applies a partitioning function to determine the next operator to which to send the data.

The 1:M BROADCAST EXCHANGE connector reads from an operator and sends the same data to a specified collection of M operators.

## 2.3 AsterixDB Runtime(s)

AsterixDB extends the language-agnostic layer provided by Algebricks to create a scalable SQL++ processor. An AsterixDB cluster configuration defines the number of nodes and the number of partitions in each node. The system supports having multiple disks on each node and any number of partitions on each disk. The hardware and workload determine how the cluster is configured. Two options are typically considered for determining the number of partitions on a node. If the process is disk-intensive, then assigning one partition per disk allows for optimal IO. The log storage should also have a separate disk. If the process is computationally intense, like interval joins, then choosing the number of partitions should be related to the number of cores available on each node. A good practice, even for computational intense workloads, to keep one disk (or disks) designated for data with a separate disk for the logs.

Hyracks data storage and processing uses the AsterixDB — provided binary representation of various data types, including an interval (see Table 2.1). The data is sharded across all partitions on all nodes in the cluster based on hashing the primary key and distributing the records evenly throughout the cluster as shown in Figure 2.4. Let's call the result of this sharding process *global partitioning*. If an operator (such as an interval join) uses partitioning to further partition the data, we call this *local partitioning*. The data does not move to another remote partition and is only partitioned to make the local processing more efficient.

Query evaluation proceeds through the usual steps. The query is parsed into an abstract syntax tree (AST) and is then analyzed, normalized, and translated into a

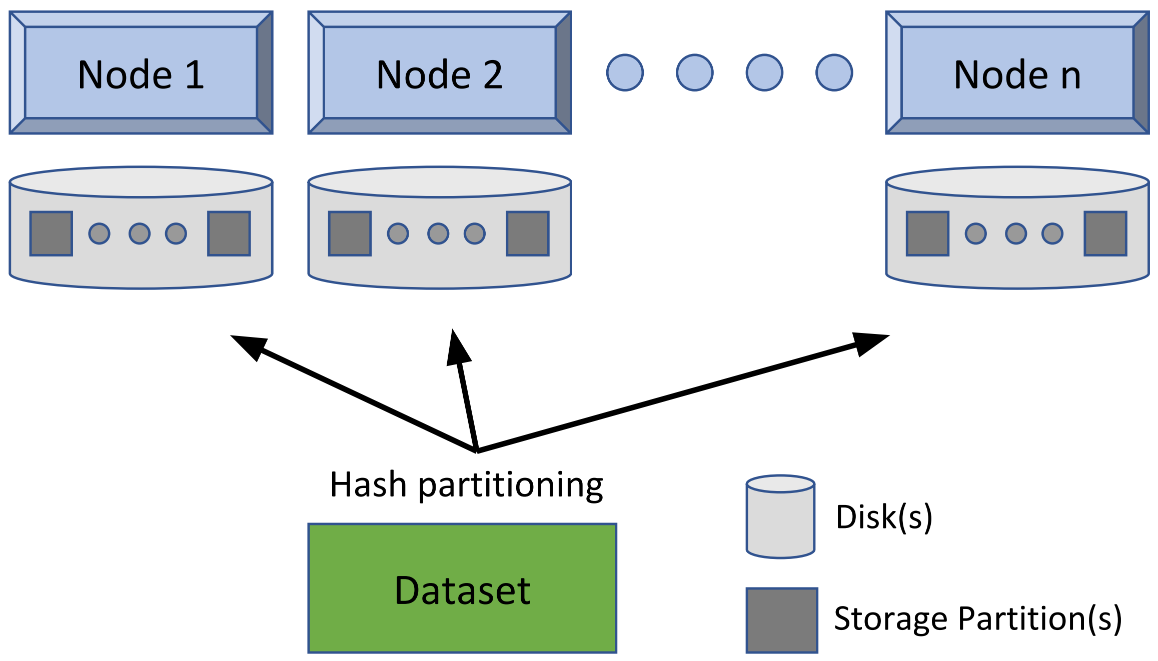


Figure 2.4: Global partitioning of a dataset in AsterixDB for interval join.

logical plan. The logical plan consists of Algebricks data-model-independent operators parameterized with Asterix data-model-dependent expressions. The logical plan is then optimized using both generic rewrite rules provided by Algebricks and AsterixDB — specific rewrite rules. After rewriting the logical plan, it is translated into a physical plan and optimized further (physical optimization includes such rules as the selection of join methods or the distribution of the plan). Finally, the physical plan is translated into a Hyracks job that is executed. Similar to Algebricks operators that have physical representations based on Hyracks operators, AsterixDB provides executable functions that implement AsterixDB's data-model-dependent expressions.

At runtime, the AsterixDB cluster processes a query that arrives via the web interface or the RESTful query API served from the cluster controller. The process starts with a user submitting a SQL++ query statement to AsterixDB for parallel execution. The cluster controller parses and optimizes the query and then submits the generated Hyracks job to the cluster controller, which manages and distributes tasks to each of the data nodes for evaluation. Each data node contains the globally hash-partitioned records as well as the AsterixDB runtime expressions used to evaluate the node's tasks. Finally, the cluster controller collects the data nodes' results and sends the result back to the cluster controller, which returns the result to the user.

## 2.4 Partitioning for Parallel Join Query Plans

The AsterixDB query optimizer is responsible for recognizing join queries and implementing the logical join operator. After the logical optimization step is finished, the physical optimizer will select the physical join operator. The physical operator describes the global partitioning and local data properties required to complete the operator's task in a shared-nothing architecture. Each physical join operator creates the join results for its partition; they are then combined together to form the complete result. The data must be globally partitioned and ordered in a way that allows the physical operator to create the complete result. Table 2.2 shows a list of example joins and their required global partitioning properties.

The physical join operator must pick a global partitioning strategy that ensures that the overall join process will yield the complete join result when all result partitions are combined. Consider the nested-loop join, a brute force join operator, which compares every tuple in the left (or build) dataset with every tuple in the right (or probe) dataset.

| **Join Name** | **Physical Operator** | **Left Dataset** | **Right Dataset** |
|---|---|---|---|
| Nested Loop | Block Nested Loop | Hash Partition | Broadcast Partition |
| Hash Join | Hybrid Hash Join | Hash Partition | Hash Partition |
| Broadcast Join | Hybrid Hash Join | Broadcast Partition | Hash Partition |
| Merge Join | Merge Join | Hash Partition, Locally Sorted | Hash Partition, Locally Sorted |
| Sort Merge Join | Merge Join | Range Partition, Locally Sorted | Range Partition, Locally Sorted |

Table 2.2: Various parallel joins and their required data partitioning strategies.

The global partitioning strategy for the physical nested-loop join operator ensures that each tuple in the left dataset can be matched with each tuple in the right dataset. The physical nested-loop join operator must be able to apply its local join process to each joined partition and guarantee that all join results will be created. That is, each tuple in the left partition must be compared to every other tuple in right partition. To scale this join approach, the system must ensure that the global partitioning strategy allows for every tuple in the left dataset is be compared with all possible matching tuples in right dataset. One way to scale this join to many nodes is to HASH PARTITION the left (or build) dataset to allow for minimal data in each join operator partition, as shown in Figure 2.5. The right (or probe) dataset is BROADCAST (which is stored using a globally HASH PARTITION strategy) to make each global partition hold the whole right dataset, as shown in Figure 2.6. While a

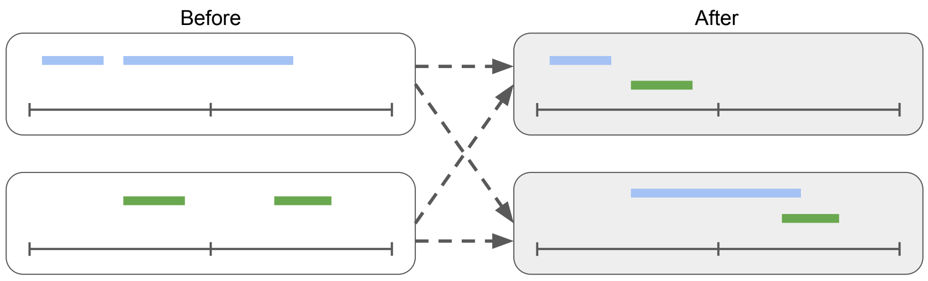


Figure 2.5: Hash partitioning applies a partition function to the data and re-partitions based on the hash results.

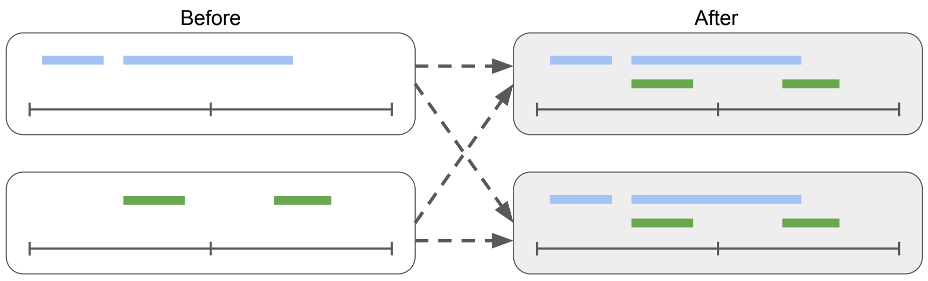


Figure 2.6: Broadcast partitioning sends each partition to all other partitions to create the full dataset in each partitions. The figure shows how the resulting partitions grows to hold the whole dataset.

copy of the right dataset is distributed to all partitions, pieces of the left dataset are spread evenly across partitions.

Using the BROADCAST connector means the right dataset is sent over the network to every other partition and each local join computation must process the whole right dataset. For join predicates that can be supported by hash join, the partitioning for the right dataset could also use HASH PARTITIONING, thus reducing the network and local processing for the join, not to mention using a faster (hash-based) local join algorithm.

Sort-merge join is an alternate approach to a hash-based join. The sort-merge join requires that the global partitions hold all tuples that could be matched and locally sorted. For data that is already locally sorted, the global partitioning strategy could either

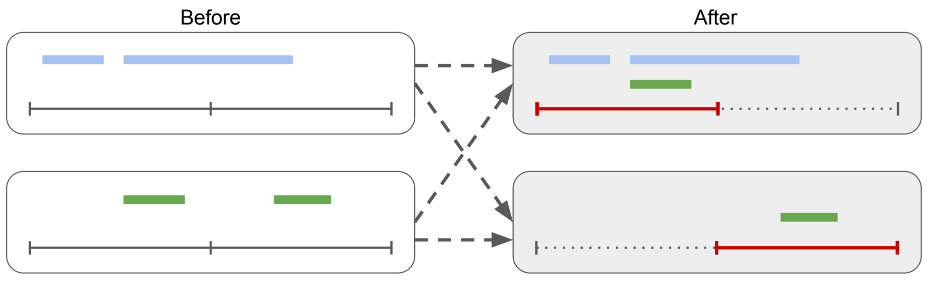


Figure 2.7: Range partitioning creates partitions where each partition holds a specific range of join key values.

be hash-partitioned or range-partitioned. Both of these global partitioning schemes ensure that the join key from each dataset will be located in same partition. Figure 2.7 shows a range partitioned layout which keeps the partitions with a global time range order as compared to hash partitioning where partitions do not have a global order.

The local join operators use different global partitioning schemes to ensure the correct overall join result. Traditional joins use the global partitioning schemes discussed in this chapter. Chapter 3 describes an additional global partitioning scheme that could be added to Apache AsterixDB to support interval joins.

# Chapter 3

# Scaling Overlapping Interval Joins

## 3.1 Introduction

For the purposes of this dissertation, time is assumed to be discrete and is described by a succession of consecutive non-negative integers. A time interval is represented by two integers: **start** and **end**, where **start** $<$ **end**. Further we assume that an interval is semi-closed as in [**start**, **end**), meaning that it contains all the time instants starting from **start**, but not including the **end** time instant. For the remainder of this dissertation we will use the terms "time instant" and "point" interchangeably. If data is too large for a node's main memory computation, it may "spill" to disk to be processed later. Such spilling can significantly impact the join algorithm's performance, especially since we do not assume the existence of any indexing or other data structures on the data to be joined.

In a shared-nothing environment, each node has a part of the data (called a data partition) and operates on it independently. The system uses local processes to execute query plans which are made up of operators and connectors. When implementing an interval join on a shared-nothing big data management system, the parallel interval join approach involves three phases of query processing: a global partitioning (where data may be sent to other nodes), a local sort, and a local join. The local phases use operators which only have access to local partitions of data. Note that there are typically two stages where data is distributed among nodes. First, there is an initial distribution of the interval data among the nodes before the query begins (i.e., for data storage); this has traditionally been done using a hashing scheme on the tuple key. Second, there is a subsequent distribution (global partitioning, sometimes called repartitioning) of interval data from the node that initially hosted the data to any other node(s) that may need the data to perform the requested join operation. (Improving this distribution approach is the focus of later discussion.) The global partitioning phase ensures that all interval data (whether from the local node or remote nodes) is grouped based on their time instances for the local join operator. Then the local sort phase proceeds by sorting each local partition's interval data. For the sorting phase, we sort intervals lexicographically based on the start point followed by the end point. Finally, the interval join is processed using sorted interval data. The distributed result operator collects all partition results from all nodes to build the complete result.

The state-of-the-art algorithms for interval overlapping joins can be divided into two categories, namely algorithms that utilize local partitioning and those that do not further partition the data; these are called *partitioning* and *non-partitioning* algorithms, respectively. The non-partitioning interval join algorithms – sort-merge (SM) (as described by [38]), time-sweep (TS) (a modified algorithm from [43]) and forward scan (FS) [16] – have been extended by us to work within a memory budget. The partitioning interval join algorithms – overlap interval partition join (OIP) [24] and disjoint interval partitioning (DIP) [17] – come from papers that included descriptions of the query processing for both in-memory and on-disk data. The on-disk descriptions from these papers are used in our work when the memory budget is insufficient for in-memory operations.

This chapter makes three contributions to overlapping interval joins: (i) we implement various state-of-the-art overlapping interval join algorithms on AsterixDB, an open-source shared-nothing big data management system; (ii) in doing that, we had to extend AsterixDB to include an interval partitioning connector and update the query optimizer to recognize and construct an interval join query plan; and (iii) we extend the non-partitioning algorithms to support "spilling", i.e., to work under a limited memory scenario.

We proceed by describing the changes applied to AsterixDB so as to support an interval join query plan in Section 3.2. Section 3.3 details how each considered algorithm is extended to work within a memory budget and its implementation in AsterixDB. The performance of the overlapping interval joins are evaluated in Section 3.4, and conclusions appear in Section 3.5.

## 3.2 Extending AsterixDB for More Efficient Interval Joins

Prior to our work, the query language of AsterixDB supported interval expressions that can be used to create an interval join query. Consider two datasets, *Staff* and *Students*, each with an interval data field. The *Staff* dataset has fields *name*(string) and *employment*(interval), while *Students* has fields *name*(string) and *attendance*(interval). In

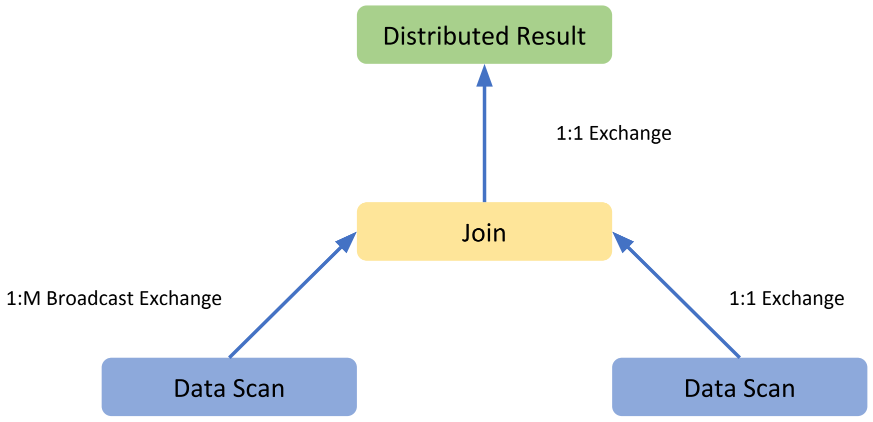


Figure 3.1: The initial query plan for Interval joins in AsterixDB.

each dataset the primary key is the field *name*. Using these datasets, a sample SQL++ interval join query is the following:

```
SELECT element{ 'staff':f, 'student':d }
FROM Staff AS f, Students AS d
WHERE interval-overlapping(f.employment, d.attendance);
```

Running this join query on AsterixDB before our work would have selected a nested-loop join operator since the optimizer did not support specific interval joins. The join operators and connectors for a simplified plan generated from this query are shown in Figure 3.1. The logical join operator uses a nested-loop physical join operator which defines the global partitioning properties needed for the join operation. With this approach, the data will be distributed among the nodes only once, before the join operation starts. One dataset (shown on the left of the figure) is broadcast so that the entire dataset exists on all partitions while the other dataset (on the right) transfers data using a 1:1 exchange which just copies its portion of the data locally to the next operator while keeping its original hash partitioning on the primary key (wherein each node contains a part of this dataset).

The initial BROADCAST exchange connector in Figure 2.6 creates multiple copies of the left dataset, one for each partition. This level of network traffic and data duplication impose a significant overhead, and the local join algorithm (a brute-force, nested loop join that compares all tuples) is not optimal. To address these inefficiencies, we implemented both a new partitioning scheme and a new join operator which we describe next.

Chawda et al. [20] describes how a generic Map-Reduce interval join can be run on multiple machines in the Hadoop world and defines how to globally partition interval joins (the map phase) for many different types of interval joins using Allen's interval algebra [2]. In particular, they demonstrated how to map data into partitions for independent local join processing with replicated interval data so as not to create duplicate join results. Three partitioning schemes were proposed (*PROJECT*, *SPLIT*, and *REPLICATE*) that each distribute the tuples based on an interval's start time and end time (so that a tuple can be distributed to multiple nodes) rather than on a hash of the tuple key. Each partition is designed to be a non-overlapping temporal range defined by a set of split points given in a query hint. *PROJECT* partitioning transfers an interval to a global partition whose temporal range holds the interval's start point (or end point, depending on the join condition) as shown in Figure 3.2. The range partitioning from Figure 2.7 can be used to perform *PROJECT* partitioning by selecting the partitioning key to be either the interval's start or end point. In contrast, *SPLIT* partitioning (partially) broadcasts an interval to all global partitions whose temporal range overlaps the interval as shown in Figure 3.3. Finally, *REPLICATE* partitioning transfers an interval to the global partition whose tem-

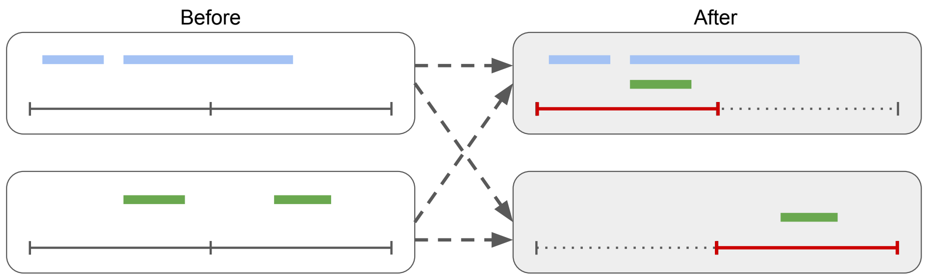


Figure 3.2: *PROJECT* partitioning is the same as range partitioning with the interval start point as the key.

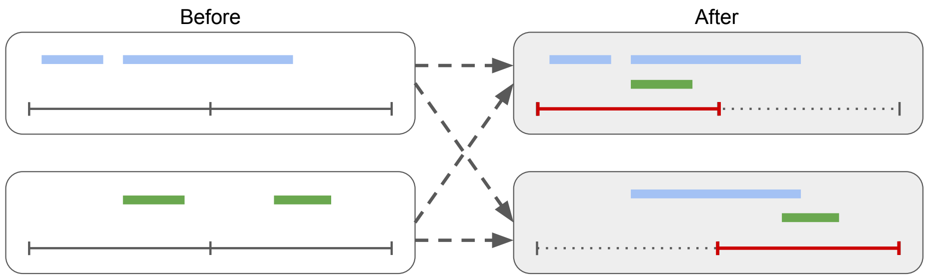


Figure 3.3: *SPLIT* partitioning is similar to range partitioning, but includes additional values that overlap multiple partitions.

poral range holds the interval's start point and then (partially) broadcasts the interval to all subsequent (later in time) partitions as shown in Figure 3.4.

The Map-Reduce interval join of [20] focuses on the global partitioning (mapping phase) of interval joins, while its local join process (reducer phase) only utilizes a nested-loop join. AsterixDB supports many traditional database operators, like aggregate and join. The

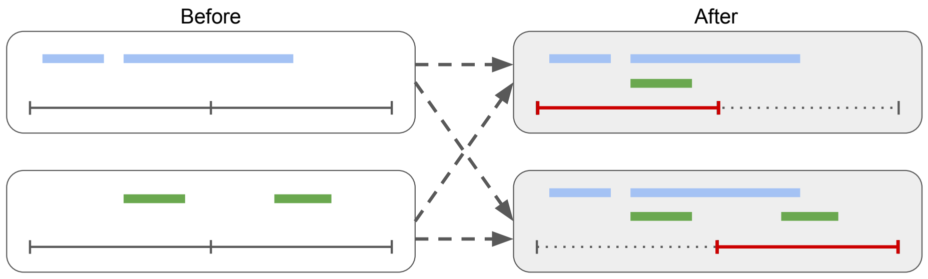


Figure 3.4: *REPLICATE* partitioning is similar to range partitioning, but partially broadcasts each interval to all partitions with its start point and subsequent partitions.

global partitioning from [20] can instead be paired with a specialized local join operator that is more focused on applying an efficient join algorithm to a local partition. Other supporting operators are discussed later in this section when we introduce an AsterixDB interval join query plan. This dissertation utilizes the global partitioning from Map-Reduce interval join but connects it with five different interval join algorithms for local join processing.

Chawda's *SPLIT* partitioning for Allen's interval algebra can be used to partition interval data for overlapping interval joins. Consider the following self-join example which uses the data from Figure 3.3. The two identical datasets have been labeled $R$ and $S$ in Figure 3.5 to show a join scenario. (The two look ”similar” because the example assumes a self-join where $R$ and $S$ are actually the same interval collection.) Each dataset has two partitions divided into two non-overlapping temporal ranges identified by the red lines. The join process needs to create the complete overlapping interval join result for the labeled intervals, which is [(a,w), (b,x), (b,y), (b,z), (c,x), (c,y), (d,x), (d,z)]. The two first temporal range partitions, the top left and top right partitions, have been assigned intervals based on the intervals' start times. The first temporal range partitions creates the following overlapping intervals: [(a,w), (b,x), (b,y), (c,x), (c,y)]. The bottom two partitions represent the second temporal range and create a few scenarios that need to be addressed by using *SPLIT* partitioning. The long interval b in dataset $R$ have been sent to both the first and second temporal ranges, similarly for x in dataset $S$. The two duplicated intervals (b and x) are needed to ensure that they will be joined with the short intervals (d and z, respectively) in the second temporal range. First, consider b; it must be in temporal range one for matching with y and x and in temporal range two for matching with z. Similarly for x, it must be in temporal range one for matching with b and c and in temporal range two for matching with d. As a result of b and x being in temporal range two, the following pairs are created: [(b,x), (b,z), (d,x), (d,z)]. Using *SPLIT* partitioning strategy ensures that local join processing can create all the interval join results. However, note that duplicating the long interval (b and x) has created two (b,x) results: one in the first and one in the second temporal range. To ensure no duplicate results, the join algorithm must not create interval pairs for intervals whose start times are before their partition's designated temporal range. In this case, the second temporal range partition would then not create the (b,x) pair and the overlapping interval join results would not have any duplicates.

Using the *SPLIT* partitioning scheme from [20], Figure 3.6 shows an updated interval join query plan. The primary-key-partitioned data is read with a DATASCAN operator and uses a 1:1 EXCHANGE to connect to the next operator. The FORWARD operator reads the split points from a query hint (defined by the query writer) and shares the split points with an M:N *SPLIT* MULTICAST EXCHANGE connector to partition the intervals. The data is redistributed using the M:N *SPLIT* MULTICAST EXCHANGE connector to put the data in an ordered partition layout based on the *SPLIT* partitioning details. Since all the local interval join methods described in this chapter require the data to be sorted before starting the joining phase, the ORDER operator has been added after M:N *SPLIT* MULTICAST EXCHANGE connector to do the required sorting on each local partition. On each node, the data is then streamed into a join operator that computes a local interval join. The result may be streamed to another node (to be merged with results

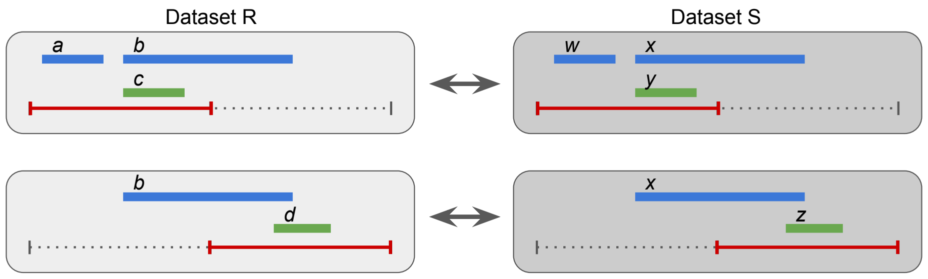


Figure 3.5: A sample join for Datasets $R$ and $S$ (where $R$ and $S$ have the same data) each with two partitions created using global *SPLIT* partitioning.

from other nodes), or maybe left on the current node for later retrieval and viewing, as determined by the distributed result operator.

Our new interval join algorithms required the addition of several new items to the AsterixDB code base. A new connector, called M:N *SPLIT* MULTICAST EXCHANGE, was implemented to apply a new *SPLIT* partition function that defines to which N operators (on the same or other nodes) the data is sent. To incorporate the range split points into the *SPLIT* partitioner, a FORWARD operator was introduced for reading the range query hint and sharing the range split points with the M:N *SPLIT* MULTICAST EXCHANGE connector.

The SM, TS, and FS algorithms require a more dynamic memory manager than existed with AsterixDB. The previous memory manager allows tuples to be added to the join operator's memory and once they had been used, the join's memory was completely wiped. These three interval join operators need the ability to add and remove tuples as needed due to the interval properties and the algorithms process. The new memory manager supports dynamically adding and removing manages tuples while minimizing garage collection. The manager also includes an iterator for processing the available tuples.

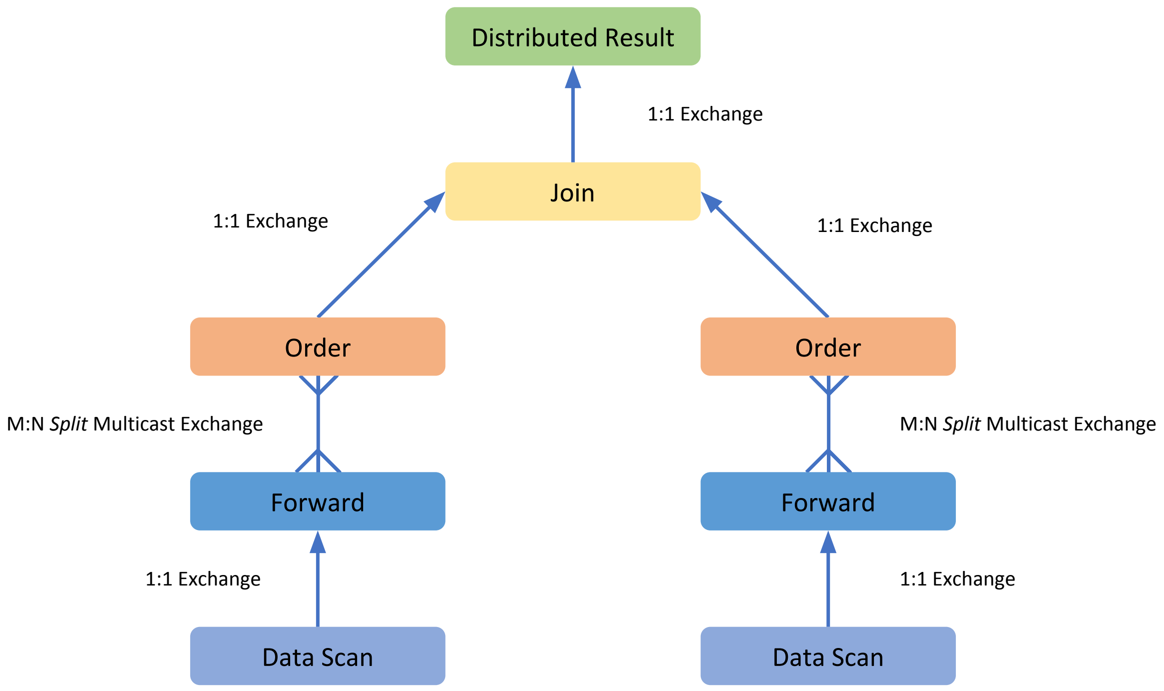


Figure 3.6: Updated query plan for AsterixDB interval join ordered partitioning and local sorting with *SPLIT* partitioning.

Finally, the five interval join algorithms utilize the previous JOIN logical operator and extend a new stream join physical operator that works at a more granular (and thus more efficient) level. Specifically, the previous join physical operators had a blocking edge between the processing of the two input join streams. As a result, all data from one dataset had to be processed (”built”) before starting the join process (”probe”). The new stream join operator is able to start the join processing with the first tuple in the data stream, that is, neither branch needs to be fully processed to start the join process. Note that partitioned interval joins still use the previous (non-steam) join approach because such joins can only occur after one side has been completely partitioned.

## 3.3 Overlapping Interval Join Algorithms

The join operator shown in Figure 3.6 can support many different local join methods. In this section, we present five options for local interval join methods: sort-merge (SM), time-sweep (TS), forward-scan (FS), overlap interval partition join (OIP), and disjoint interval partitioning (DIP). Each method has been implemented in AsterixDB, optimized in order to scale, and runs within a memory budget. For the following algorithmic descriptions, consider an overlapping interval join between two datasets: $R$ and $S$. For each algorithm, we will describe the in-memory method and then explain how, when memory is full, the algorithm completes the join using a spilling phase.

### 3.3.1 Sort-Merge Interval Join

An interval join algorithm for parallel processing was first defined by Leung and Muntz [38]. They defined a three phase (replication, join, and merge) process for doing a join on interval data that works well in a shared-nothing environment. The Leung and Muntz algorithm was designed only for a single machine and did not consider limited memory. Their replication phase is similar to our *Split* partitioning, while the join and merge phases resemble the local sort-merge interval join operator we discuss in this section. Instead, our SM algorithm scales to many nodes, works within a memory budget, and does not need to remove duplicates after the join due to the partitioning scheme used before the join.

Note that the behavior of a sort-merge interval join is similar to a traditional sort-merge join when many duplicate keys are present. The difference is that while duplicate keys in a traditional sort-merge will only match with the same key, in interval joins each

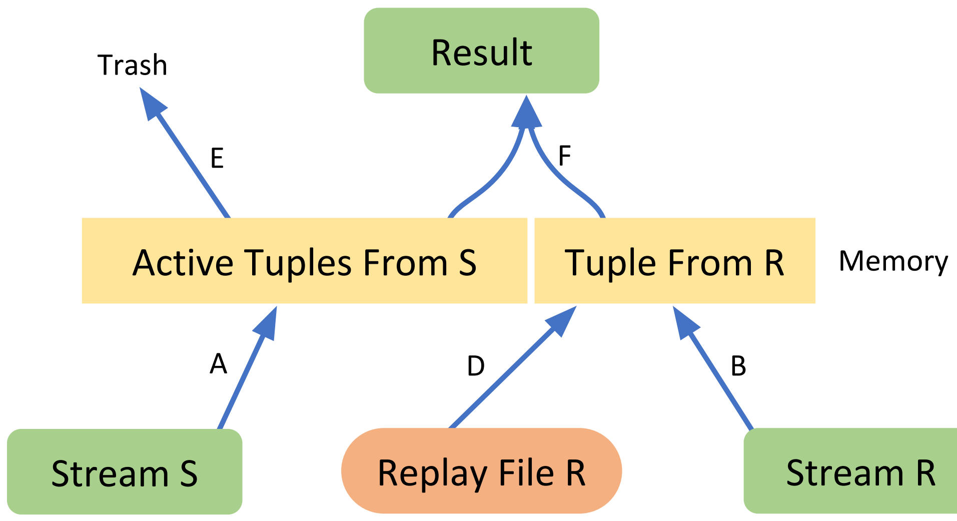


Figure 3.7: Sort-merge interval join actions for the in-memory mode. A) move a tuple from the stream of $S$ to memory; B) move a tuple from the stream of $R$ to memory; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; and F) put the merged the tuples from $R$ and $S$ in to the result stream.

interval matches with all tuples with overlapping intervals. The basic idea for SM is to sort the two datasets and then merge the result by picking an interval from dataset $R$ and testing it for overlapping tuples in $S$. The process is repeated for every tuple in dataset $R$. Since the tuples are in sorted order, the matching process does not need to scan the whole dataset $S$, instead scanning only the range of tuples that are in the overlapping area. The algorithm has a low number of extra comparisons since the algorithm stops processing an interval $t_R$ once the process finds a interval $t_S$ that starts after $t_R$ ends.

The SM algorithm requires both dataset streams to be sorted by the interval start-point followed by end-point. First, a single tuple ($t_R$) from $R$ is loaded into memory from the stream, as shown by the B arrow in Figure 3.7. Tuples from the $S$ stream are loaded into memory (as shown by the A arrow in Figure 3.7) and, as each $t_S$ tuple is loaded into memory, the tuple $t_R$ in memory is compared to the interval of $t_S$ and, if they overlap,

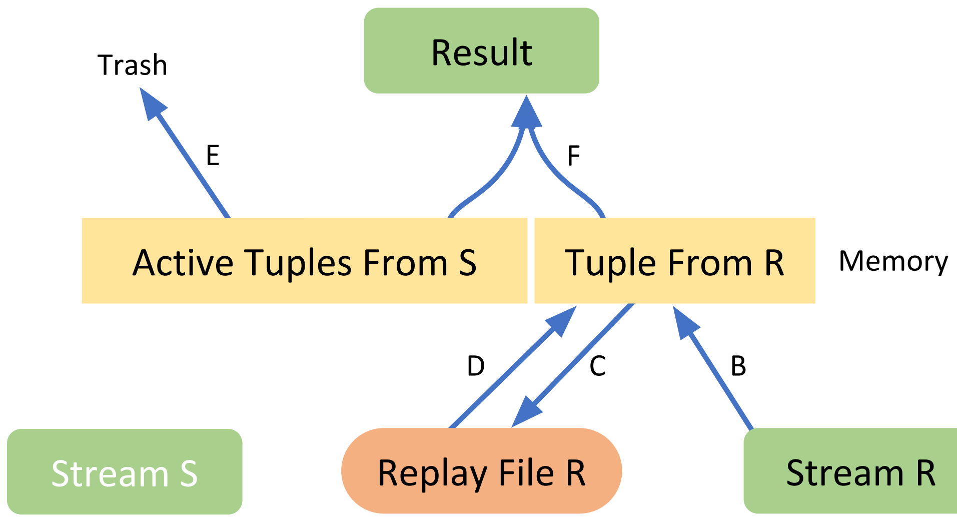


Figure 3.8: Sort-merge interval join actions for the spilling mode. B) move a tuple from the stream of $R$ to memory; C) copy the $R$ tuple from memory to the spill file for $R$; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; and F) put the merged the tuples from $R$ and $S$ into the result stream.

the join result is produced as shown by the F arrow in Figure 3.7. Once a new tuple ($t_S$) is loaded into memory that does not overlap with $t_R$, the loading stops since no more tuples from $S$ will match with the tuple $t_R$ in memory due to data sorting. The process repeats by loading the next tuple from $R$ into memory. Each interval in memory from $S$ is compared with the new tuple $t_R$. If $t_S$'s end-point is before $t_R$'s start-point, then it is purged from memory, as shown by the E arrow in Figure 3.7. If $t_S$ overlaps $t_R$, a join result is created. After going through all memory tuples, new tuples from $S$ are added to memory and compared with $t_R$. The process continues for all the tuples in $S$. Once all tuples in $R$ have gone through this process and been compared with tuples from $S$ that are in memory, the join is complete.

The SM algorithm may run out of memory if $t_R$ is overlapping with more tuples from $S$ than can fit in memory (when there is no more space to hold a tuple in Active Tuples

From S). When memory is used up, the algorithm can no longer continue processing the join in memory. In this case, the memory used for active tuples from $S$ could overlap with future tuples from $R$ so they cannot be permanently eject from memory. The spill algorithm will start by continuing to load individual tuples from $R$ (as shown in the B arrow in Figure 3.8) and compare them with the active tuples from $S$ in memory. Since these tuples from $R$ may overlap data in the $S$ stream, these tuples will be written to a replay spill file, as shown by the C arrow in Figure 3.8. No new tuples from $S$ are added to memory during this process, but tuples from $S$ are purged when they no longer match with $R$ (as shown in the E arrow in Figure 3.8). The process continues until all active tuples from $S$ have been removed from memory. At this point, the in-memory join method may resume with one condition. While $S$ will stream tuples into memory, dataset $R$ must start by loading tuples from the replay spill file (as shown in the D arrow in Figure 3.7) and then continue into the $R$ stream. The process repeats each time memory is full, and the $S$ stream is paused to free up memory.

### 3.3.2 Time-Sweep Interval Join

Instead of scaling up resources (more memory per node or more nodes), another approach is to speed up interval join queries through the use of an index. For example, [34] uses the Timeline Index, while [25] uses the RI-index for the interval join; however, these indexing methods are created before executing a join. This requirement is unattractive because we are looking for algorithms that can run ad hoc interval joins on AsterixDB without any changes to existing stored datasets. Instead, Piatov [43] defines an end-point based interval join algorithm, that uses a sweep-based approach focused on compact memory management to maximize cache utilization. The algorithm runs in memory and requires an in-memory index that can be built on-the-fly. The algorithm builds the end-point index in memory and then uses a second pass to perform the join using the index. [43] outlines a few optimizations to minimize cache misses during the joining phase. Our time-sweep algorithm alters the end-point interval join of [43] in two ways, so that it can be performed in a single pass when the memory budget is sufficient and so that it also works when memory is limited.

Like most of the other interval join algorithms, this algorithm requires interval data to be sorted by [begin | end] point before starting the time-sweep interval join. Our implementation has a few key differences. In particular, our algorithm utilizes the incoming sorted interval data as the index for start-points and only builds a simple min heap for the end-points while they are stored in memory, shown as ”Delete Order” in Figure 3.9. The Delete Order data structure is also used to speed up memory clean up which incorporates the optimizations discussed in [43]. Doing this allows the join process to only perform one pass over the dataset. The algorithm has also been extended to work when memory is limited by using a similar process to SM to up free memory.

The algorithm starts with the first interval in time order from either dataset. Assume that $R$ has the first time-ordered interval $t_R$. The interval is added to $R$'s active tuples in memory (as shown by the B arrow in Figure 3.9), and the interval end-point is added to the Delete Order data structure structure. The next operation is based on where the lowest remaining end-point is found, which may come from the Delete Order, stream $R$, or stream $S$. If the interval is from dataset $S$, it is added to memory (as shown by the

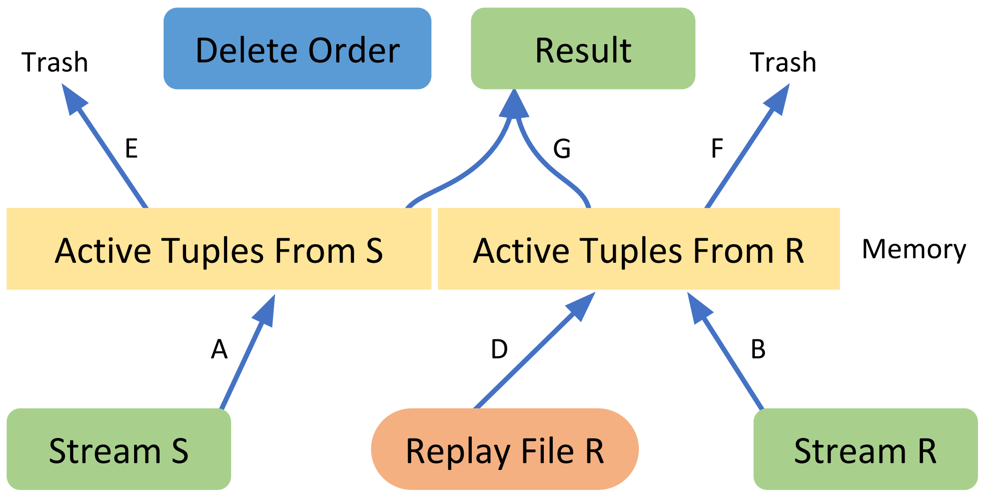


Figure 3.9: Time-sweep interval join actions for the in-memory mode. A) move a tuple from the stream of $S$ to memory; B) move a tuple from the stream of $R$ to memory; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; F) remove an active tuple in memory from $R$; and G) put the merged the tuples from $R$ and $S$ into the result stream.

A arrow in Figure 3.9), joined with all active tuples in memory from $R$ (as shown by the G arrow in Figure 3.9), and added to the Delete Order. The reverse is true for adding a tuple from dataset $R$'s stream. If the next time ordered end-point is from the Delete Order, then the tuple linked to that end-point is removed from memory, as shown by the E or F arrow in Figure 3.9. Tuples are added and removed so that only tuples that hold active intervals during the time sweep are in memory. No additional comparisons are needed as the time-sweep algorithm properties ensure that they are overlapping.

If the number of active tuples from both datasets exceeds the available memory, the algorithm must stop the in-memory join. In an effort to free the most memory, the algorithm picks the memory partition with the most active tuples. As an example, assume that $S$'s memory partition has the most tuples. Since the $S$ active tuples in memory may match with future tuples from $R$, all active tuples from $S$ must be joined with $R$'s data

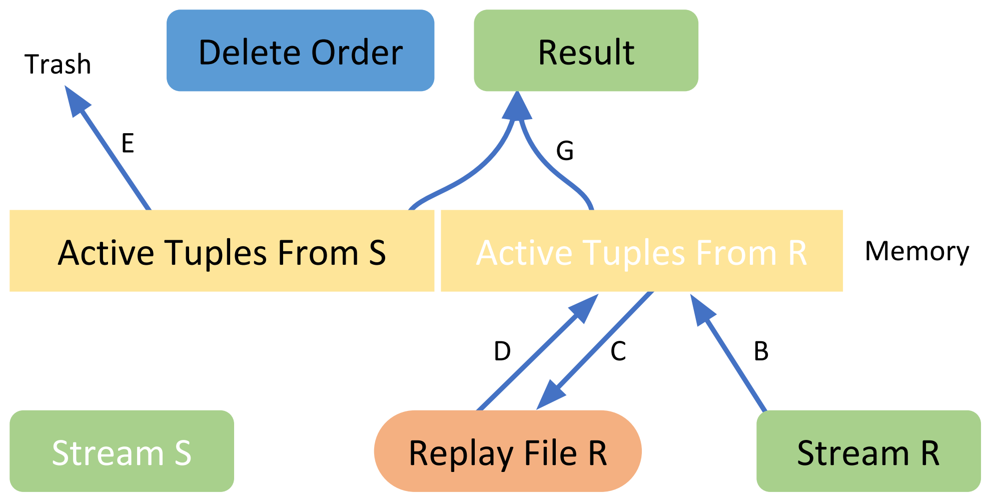


Figure 3.10: Time-sweep interval join actions for the spilling mode. B) move a tuple from the stream of $R$ to memory; C) copy the $R$ tuple from memory to the spill file for $R$; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; and G) put the merged the tuples from $R$ and $S$ into the result stream.

stream. The $R$ data stream is processed tuple by tuple (as shown by the B arrow in Figure 3.10), comparing tuples with $S$'s active tuples in memory. Similar to the SM process of freeing memory, $R$'s data stream is written to a replay spill file (as shown by the C arrow in Figure 3.10), and $S$'s active tuples in memory that no longer match are removed both from memory and the Delete Order, as shown by the E arrow Figure 3.10. Once $S$'s memory partition is emptied, memory has been freed and the in-memory join can resume. The algorithm will continue to load tuples from the $S$ stream and start with the $R$ replay spill file and then continue with $R$ stream, as shown in Figure 3.10 D and B, respectively. While Figure 3.10 only shows a replay file for $R$, the spill process could be conducted for either dataset, depending on which dataset has more active tuples in memory.

### 3.3.3 Forward-Scan Interval Join

The Forward-Scan Interval Join [16] is based on the classic Plane Sweep Algorithm [44] and takes a different approach than the time-sweep interval join [43]. Forward-Scan showed how to scale up to many threads on a single machine using several different partitioning strategies. For this dissertation we focus on implementing the Forward-Sweep local join process in AsterixDB (a multi-node shared-nothing database system) and extend it for working within a memory budget.

The Forward-Scan join performs a sweep through the sorted (by start points) intervals dataset. The algorithm continues to pick the interval with the next start point, matches it with all overlapping intervals in the other dataset, and then the initially-selected interval is removed from memory. The process repeats, picking the interval with the next start point and matching all overlapping intervals in memory. The process first joins tuples in memory and then loads any new tuples, as needed from the stream, to complete the join.

Consider two datasets $R$ and $S$ where $R$ has the first time-ordered interval $t_R$. The interval $t_R$ is added to $R$'s active tuples in memory, as shown by the B arrow in Figure 3.11. The active tuples from $S$ are scanned to find all overlapping tuples with $t_R$ and added to the result set. Tuples from stream S are added to memory (as shown by the A arrow in Figure 3.11) and matched with $t_R$ until a $t_S$ tuple is found that starts after the $t_R$'s interval ends. If during the scan of the active tuples from $S$, a tuple $t_S$ is found that does not match with a future tuple from $R$, it is removed from memory as shown by the E arrow in Figure 3.11. Once tuple $t_R$ has been matched with all overlapping tuples, $t_R$ is removed as shown

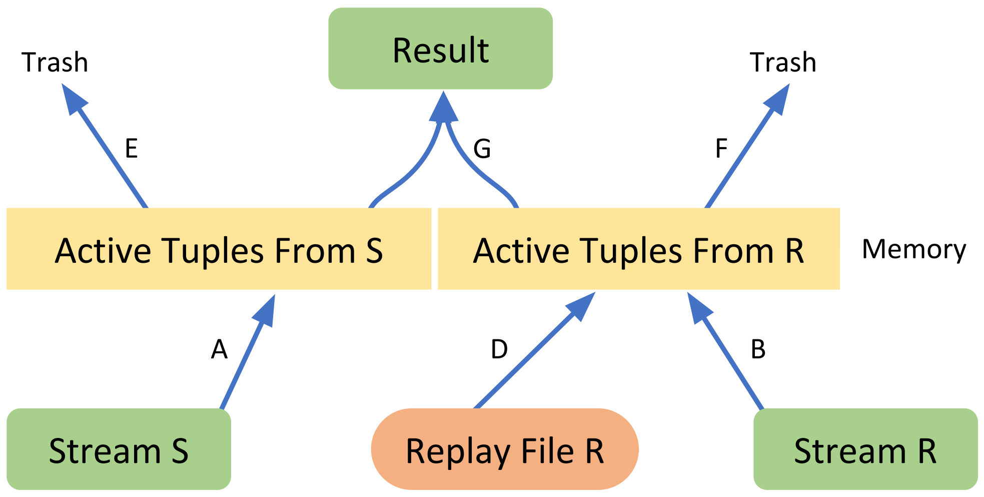


Figure 3.11: Forward-Scan join actions for the in-memory mode. A) move a tuple from the stream of $S$ to memory; B) move a tuple from the stream of $R$ to memory; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; F) remove an active tuple in memory from $R$; and G) put the merged the tuples from $R$ and $S$ into the result stream.

by the F arrow in Figure 3.11. The next tuple in time order is picked (from either dataset). The process continues until all tuples have been loaded from both streams.

The join process collects active tuples through loading future tuples that overlap the tuple being processed into memory. If the number of active tuples from both datasets is going to exceed the memory budget, the algorithm must stop the in-memory join. In this case, the algorithm picks the dataset with the most active tuples to free up the maximum amount of space in memory and allow the join to continue. The spill process takes over and pauses the in-memory time-sweep algorithm. The process used to free memory is same as for the TS algorithm and can be used for either dataset, depending on which dataset has more active tuples in memory. One difference for FS is that there is no Delete Order data structure to be updated during the spill process.

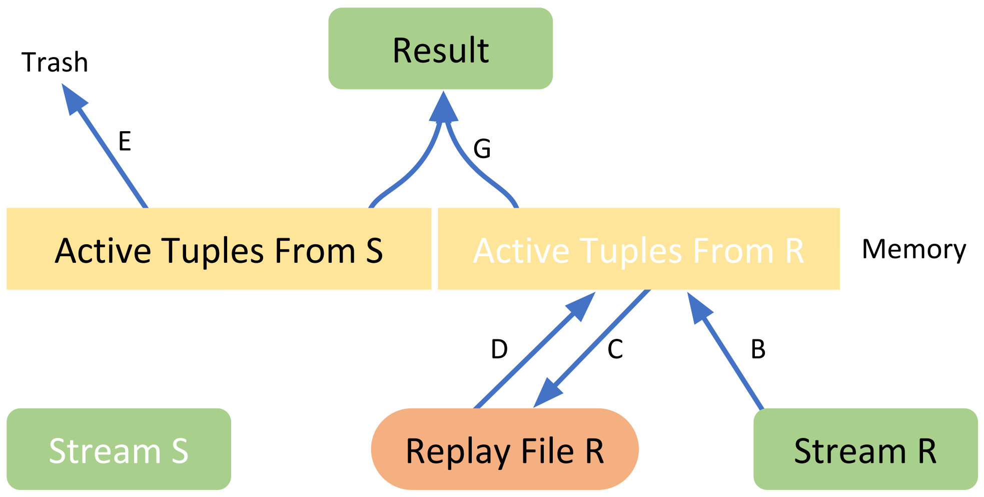


Figure 3.12: Forward-Scan join actions for the spilling mode. B) move a tuple from the stream of $R$ to memory; C) copy the $R$ tuple from memory to the spill file for $R$; D) move a tuple from the spill file of $R$ to memory; E) remove an active tuple in memory from $S$; and G) put the merged the tuples from $R$ and $S$ into the result stream.

### 3.3.4 Overlap Interval Partition Join

The non-partitioning interval join algorithms all generally have similar memory management, while the partitioning algorithms' memory management approaches are quite diverse. Overlap Interval Partition Join (OIP) [24] outlines how to create temporal partitions and perform an overlapping join. This partitioning method groups intervals with a close starting point and similar duration into separate local partitions. OIP focuses on a join process that is efficient for interval data with a few long duration intervals among many short intervals and that works in memory or on disk. The paper claims that the algorithm can outperform other disk-resident interval join algorithms due to the way that its join process incorporates statistics on the machine's CPU and I/O. The algorithm takes a set of parameters (the longest expected interval, the number of tuples in each dataset, the speed of the CPU, and the speed of an I/O request) which are used to determine the optimal number of partitions to be used during the join. OIP is the only interval join algorithm discussed in this dissertation that requires additional parameters besides the range partition split points given in the query hint.

Prior to the join process, OIP evenly splits the temporal range into slots. The number of slots is based on a formula using the longest expected interval, the number of tuples in each dataset, the speed of the CPU, and the speed of an I/O request. The OIP uses these slots to create an overlapping temporal partitioning where each partition is defined by its start and end slot. Consider a case where three slots are used for partitioning, Figure 3.13 shows the temporal partitions with their identifiers, composed of the start slot followed by the end slot. The algorithm begins by partitioning the data into these temporal partitions. The partitioning process maps an interval's start point to a slot and the end point to a slot which together determines the interval's overlapping temporal partition. Then, a nested-loop join is performed for overlapping temporal partitions to produce the final result. The algorithm uses a nested-loop join for joining partitions since, if partitioned well, most of the intervals will match and be in the result. Since our queries are executed in a multi-node environment using *Split* partitioning, the first and last slots were overloaded with intervals that either start before or after the temporal range for this join operator. Thus, we added two special slots to create separate partitions for these intervals that extend beyond the nodes' responsible temporal range: one for intervals with start points before the range and a second one for intervals with end points after the range.

Sorting on the tuples assigned starting slot followed by ending slot results in grouping the overlapping temporal partitions together. Figure 3.14 shows how tuples from the

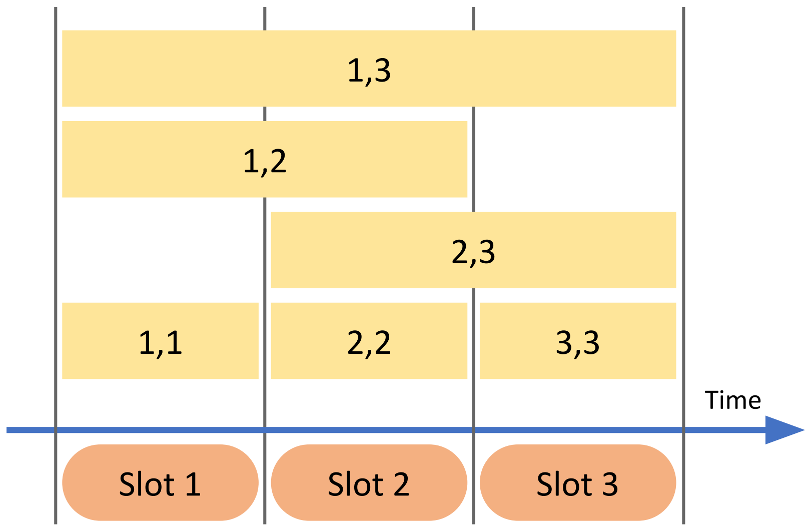


Figure 3.13: Temporal partitioning with partition identifiers when using three slots.

stream $R$ (A arrow) are added to memory. Once the interval's temporal partition is determined, the interval is written to a partition file (as shown by B arrow in Figure 3.14) and the partition count is updated. If this is a new temporal partition, the file's location for the partition start (as shown by the blue triangles in Figure 3.14) is saved in the partition locations data structure. The partitioning process is complete when all intervals from the stream have been written to the partitioning file. The process is repeated for dataset $S$.

After each dataset has been partitioned in this manner; the join uses the partition counts to calculate all overlapping temporal partitions to be joined. The join starts by reading the first tuple $t_R$ from the partition file of $R$ into the active tuples from $R$, as shown by the B arrow in Figure 3.15. Using the precalculated list of join partitions, all relevant partitions from $S$ are loaded into memory as shown by the A arrow in Figure 3.15. The partition order is maintained to ensure that the join can be completed with a single scan of partition file $S$ with the partition represented by $t_R$. The $t_R$ tuple is matched with

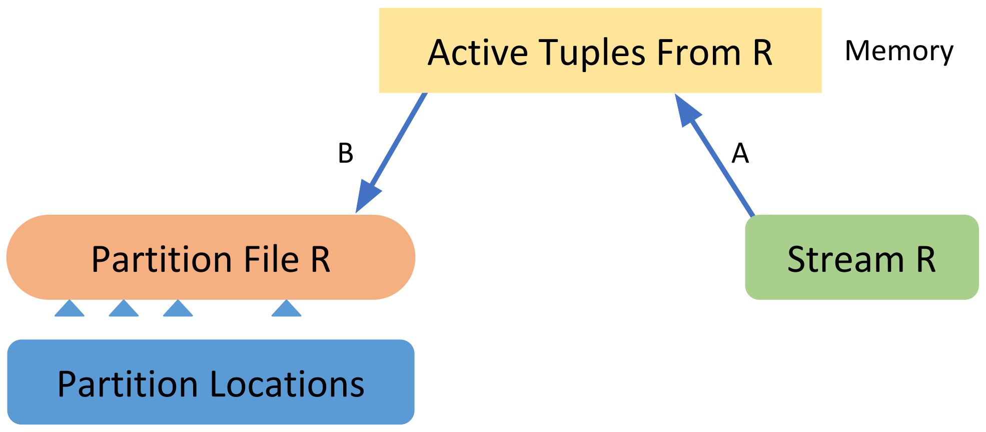


Figure 3.14: Overlapping interval join activities during partitioning. A) Move a tuple from the stream of $R$ to memory; and B) move a tuple from the memory to a partition file of $R$.

all active tuples for $S$, creating results for overlapping intervals, as shown by the C arrow in Figure 3.15 The process repeats for each tuple in $R$'s partition until all its tuples have been joined. Once the $R$ partition has completed all overlapping partition joins, the next partition in $R$ loads its first tuple into memory and the process repeats.

When dataset $S$ is larger than memory, the join will no longer be able to calculate all results using a single pass of dataset $R$. As a result, the join begins by loading as many tuples from overlapping partitions from $S$ into memory, then a single scan of the $R$ partitions is done to complete the join process all $S$ tuples in memory. The join continues by loading the next set of overlapping partitions from $S$ into memory and then joins with a single pass over $R$. The process basically becomes a block nested-loop join, but the join only needs to compare temporal partitions that are overlapping. Finally, note that $S$ will be processed in one pass while $R$ will be loaded as many times as necessary for the overlapping partitions.

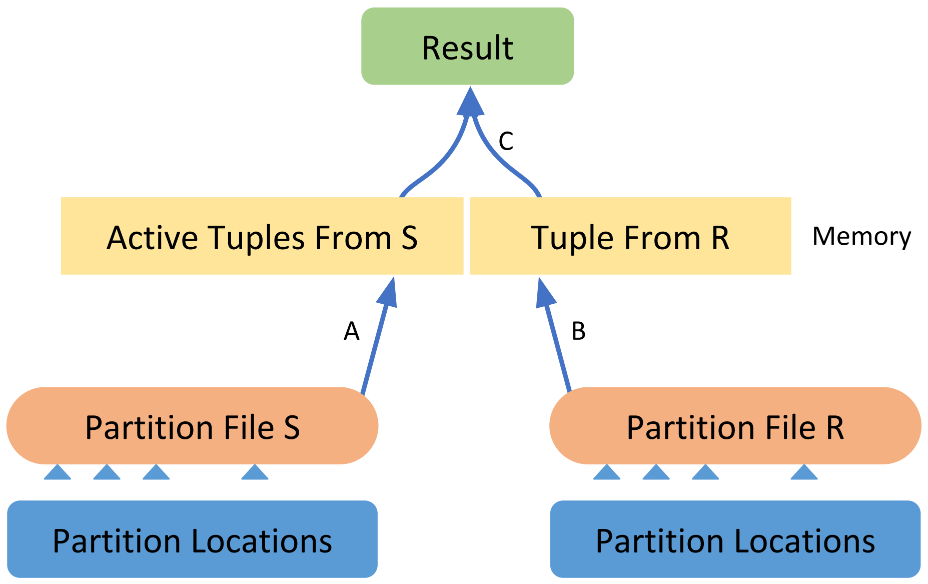


Figure 3.15: Overlapping interval join actions for the joining. A) move a tuple from the partition file of $S$ to memory; B) move a tuple from the partition file of $R$ to memory; and C) put the merged the tuples from $R$ and $S$ into the result stream.

### 3.3.5 Disjoint Interval Partitioning Join

Disjoint Interval Partitioning (DIP) [17] ensures a simple partition merge join without backtracking like in sort-merge join. The number of disjoint interval partitions needed is limited to the largest number of intervals that are active at the same time. The limit becomes a constant factor used by the join's cost model to give an upper bound on the number of data scans used during an interval join. The algorithm can be done in memory or from disk and uses two activities: one for partitioning and one for doing a merge join. The merge join process here has an advantage of only reading partitions in sequential order.

The algorithm's partitioning process begins by loading the first tuple ($t_R$) from stream $R$ into memory as shown by the A arrow in Figure 3.16. $t_R$'s partition is determined by finding a partition that does not create overlapping intervals within the partition. A list

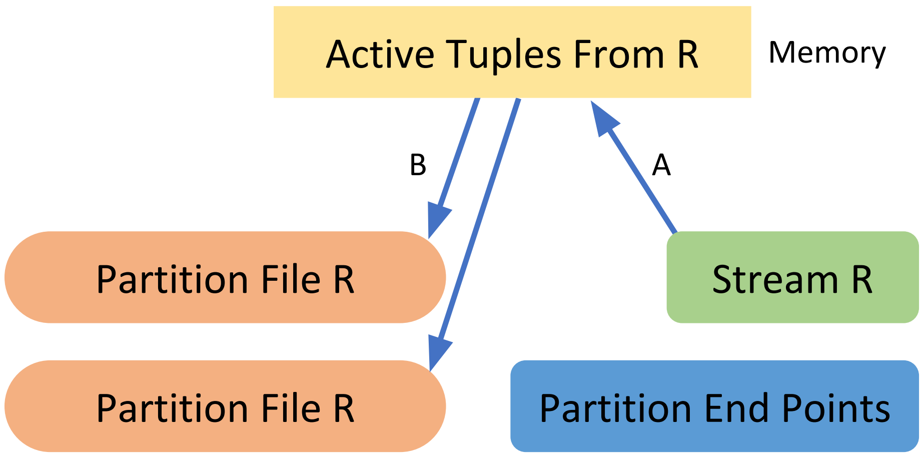


Figure 3.16: Disjoint Interval Partition Join activities during partitioning. A) Move a tuple from the stream of $R$ to memory; and B) move a tuple from the memory to a partition file of $R$.

of partition end points is saved in decreasing order, and the partitioner simply picks the next partition based on the lowest interval end-point. $t_R$ is then moved into that partition file for $R$ as shown by the B arrow in Figure 3.16, and the Partition End Points data structure is updated with $t_R$'s end point for the partition that it was added to. If $t_R$ overlaps this lowest end-point, then all existing partitions will overlap and a new partition is created. The process continues loading new stream $R$ tuples and processing them until all intervals have been assigned a partition. The partitions for dataset $R$ are written to disk and then the partition process is repeated for to dataset $S$.

Once both datasets have been partitioned, the merge join starts by picking a single partition from $R$ and merge-joining this partition with every $S$ partition. The first tuple $t_R$ is loaded into memory from an $R$ partition, as shown by the B arrow in Figure 3.17. All partitions for $S$ are loaded into memory as shown by the A arrow in Figure 3.17. The first tuple $t_R$ is checked against the first tuple for each partition from $S$ in memory. If tuple

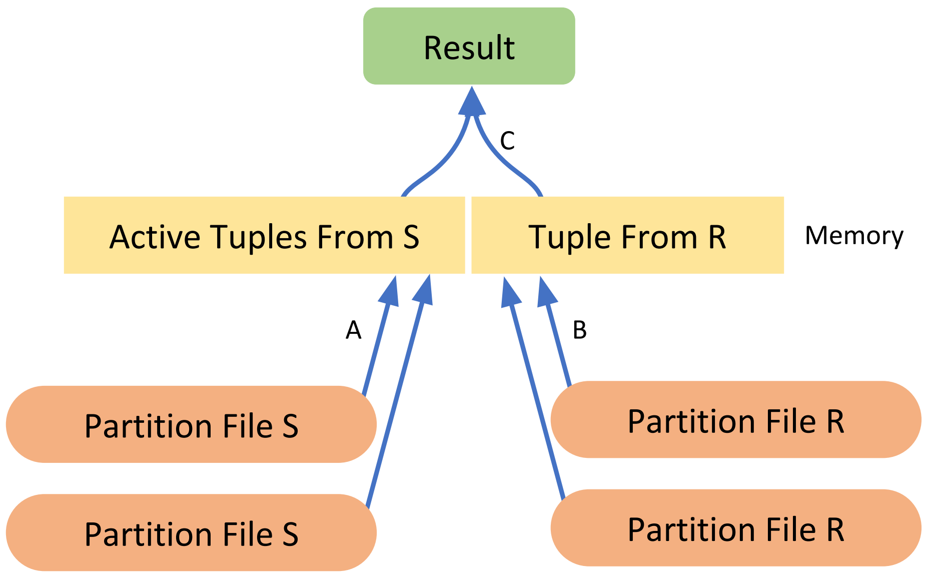


Figure 3.17: Disjoint interval partition join actions for the joining. A) move a tuple from the partition file of $S$ to memory; B) move a tuple from the partition file of $R$ to memory; and C) put the merged the tuples from $R$ and $S$ in to the result stream.

$t_R$ is after $t_S$, then the pointer for the $S$ partition is advanced. If they are overlapping, then they are sent to the result as shown by the C arrow in Figure 3.17 and the pointer for the $S$ partition is then advanced. This continues until the tuple $t_S$ start point is later then $t_R$'s end point. Then the next partition in $S$ is selected and the process repeats for each partition in $S$. After merging $t_R$ all $S$ partitions, the next tuple in $R$ is selected and merged with every partition of $S$ starting at the partition marker. The process continues until all of the tuples in $R$ have been processed or the partition marker is at the end of each $S$ partition. The next partition in $R$ is selected and its first tuple $t_R$ is loaded into memory, the $S$ partition markers are reset to the beginning of the partition and the merge process is repeated. The join is complete after processing all partitions of from $R$ with the $S$ partition in memory.

If dataset $S$ is larger than memory, the partition and join process is altered to manage memory. During partitioning, instead of keeping the whole partition in memory, only the last frame is kept in memory while the other frames are written to disk. Using one frame per partition maximizes the number of partitions (M - 1) that can be processed. A single frame is used to track all intervals that could not be assigned an overflow partition. Once all intervals have been assigned a partition or added to the overflow partition, all partition frames are flushed to disk and the process is repeated for the remaining tuples in the overflow partition. The process is repeated as many times as necessary until all tuples added to the overflow partition have been assigned a partition.

When limited memory exists during the merge join activity, batches of $S$ partitions in memory will be processed at one time. The algorithm will need multiple passes over the partitions in $R$ to complete the join. The work is now split up into sets of partitions which can be processed together by filling the available memory (M - 1). Each $S$ partition set should only be scanned once for each $R$ partition that is being joined. A page of memory is designated for the first $R$ partition, and the remaining memory pages are devoted to sets of $S$ partitions. The first tuple in $R$ will be merge-joined with the first tuples in the first frame of the $S$ partitions loaded in the first memory batch. The merge process continues until all of the $R$ partitions have been processed or the partition marker is at the end of each $S$ partition. Then the next partition from $R$ is loaded, and the $S$ partition pointers are reset to the beginning of their partition. The process continues for all $R$ partitions. Once they are complete, the next block of $S$ partitions are loaded, and the process is repeated for all $R$ partitions.

An important distinction among the two partitioning algorithms is due. In DIP, each algorithm created partition is written in a separate file (for example, Figure 3.15 shows two such partition files for $R$). Instead, OIP writes all algorithm partitions in a single partition file (see Figure 3.14). This is possible for OIP, since it first determines the number of slots, and sorts the tuples based on their starting and ending slots. Tuples are written to the partition based on this sort order. In contrast, in DIP, a tuple is assigned on the fly to one of the active partition files, based on the tuple's overlapping with the active partitions. If it overlaps with all active partitions, then a new partition file is created for this tuple. Important here is that we do not know when a partition is done (neither the size of a partition) until the whole relation is considered. Hence the DIP algorithm needs to create a separate file per partition. Based on the tuple interval characteristics, the number of partition files created can become large.

As we will see in the experimental evaluation, a large number of partition files can directly affect the DIP algorithm's spilling performance in two ways. First, during partitioning, each partition file requires a separate frame in memory; if not enough memory is available, partitioning will take extra passes. Second (and more important), during joining, each partition file has to be joined with all partition files from the other relation.

## 3.4 Performance

We have conducted a set of performance experiments to look at how these five algorithms perform in a real database system, AsterixDB. Three sets of experiments review different aspects of measuring their performance: speed-up (when sufficient memory is available), scale-up (increasing data size and available resources), and scale-up with spilling (when limited memory requires the use of disk space). All of the experiments were run on an eight-node gigabit-connected cluster. Each node has two dual-core AMD Opteron(tm) processors, 8GB of memory, and two 1TB hard drives. AsterixDB was configured to use one drive for data and one for logging. The number of partitions per node was defined in each experiment.

When preparing our experiments, we attempted to compare our performance to that reported in previously published papers. However, those algorithms were mostly implemented with bare minimum C++ code and only worked with collections of 64-bit integers. Their custom custom code was essentially a count query for an interval join. We found that such a custom-built application cannot be directly compared to a database implementation. To show these differences, we also set up a simple one-node experiment to sort one million random 64-bit integers using direct C++ code vs. using a database system.

The baseline experiment that used the C++ sorting code that was included in the Forward Scan algorithm, and it timed the process for sorting one million 64-bit integers using native C++ functions and data. Next, the same sorting process was applied to two database applications on the same list of random integers: a popular SQL database, PostgresSQL, written in C++, and AsterixDB, which is written in Java. Table 3.1 shows the times for sorting under each of these scenarios. The C++ library is roughly 14 times faster than either of the databases. The AsterixDB time is fairly close to the PostgresSQL time even though AsterixDB was written in Java. The experiment shows that the data

| Method | Language | Time |
|---|---|---|
| Vector Library | C++ | 67 ms |
| PostgresSQL | C++ | 890 ms |
| AsterixDB | Java | 914 ms |

Table 3.1: Sorting using different libraries and systems.

generality and overhead of handling full records in a DBMS does this with a performance price.

Since interval join queries tend to produce a large number of results, the experiments reported have been designed to measure the join execution time on a query with human-readable results. Figure 3.18 shows six variants of interval join queries where only the number of results returned has been changed. The query time is shown for the six different result sizes using a log scale. The first *count only* query uses an aggregate count to return the number of joined intervals. The *empty result* query creates all the interval joined pairs, but immediately after the join operator a filter has been added that will always be false, and thus no results will be returned. The *top-k* queries limit the result to k interval pairs by picking the interval pairs with the most overlap. The top-k queries include a constant time operation for each tuple to determine the interval overlap. The *full result* query returns all of the interval-joined pairs. Its time includes the cost of result generation, but does not include the time to download the result from the cluster. The *count only* query is the fastest due to only having to perform the join and apply no additional logic besides counting. The *top-k* queries are each slower than the *empty result* query due to the calculation of the interval overlap. The sorting for the *top-k* queries starts to impact the performance after exceeding 1,000 result reads. Since any small value of k has similar performance, we will choose k = 100 going forward since a person (an analyst) would be able to consume the results.

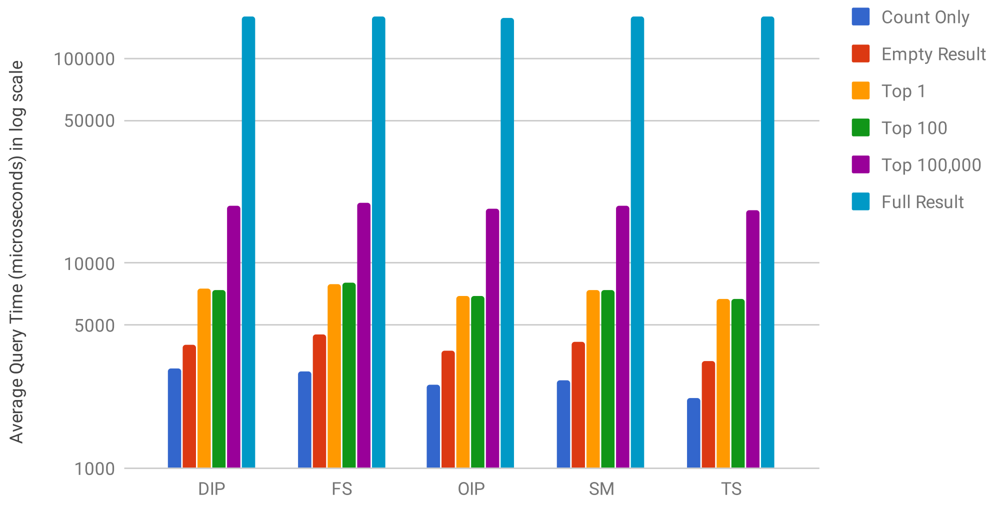


Figure 3.18: An overlapping interval join query on a single partition with various result sizes.

The final interval join query, full result, creates a new record using id, interval, and filler fields from each dataset. The filler field is used during the experiment to change the size of the tuple being processed in the join operator. An ORDER BY and LIMIT limit the results to the top 100 join results with the largest overlap. The actual SQL++ interval join query is the following:

```
SELECT element{ ‘id1’:ds1.id, ‘id2’:ds2.id,
  ‘interval1’:ds1.interval, ‘interval2’:ds2.interval,
  ‘filler1’:ds1.filler, ‘filler2’:ds2.filler}
FROM Dataset1 AS ds1, Dataset2 AS ds2
WHERE interval_overlapping(ds1.interval, ds2.interval)
ORDER BY duration_from_interval(
  get_overlapping_interval(ds1.interval, ds2.interval)) DESC
LIMIT 100;
```

The performance section continues by showing how the different algorithms speed-up with sufficient memory (non-spilling experiments). The speed-up section considers the number of partitions both on a single node and for many nodes. The scale-up experiments demonstrate the ability to add more data and corresponding processing with consistent performance. The last experiment reviews the case when memory is not sufficient and the system must spill to disk during the join.

### 3.4.1 Speed-Up Non-Spilling Experiments

For non-spilling experiments, we looked at how the queries speed-up locally using both synthetic and real data. We also considered speed-up experiments in a multi-node cluster using a synthetic dataset. We use the synthetic datasets to explore how the algorithms are affected by controlling one property of the data: cardinality, duration, or record size. The cardinality experiment represents changing the number of tuples in the dataset, by changing the intervals' "arrival rate" ($\lambda$). Changing the cardinality will affect the number of tuples in an interval's overlapping region. Similarly, changing the duration of a query will affect the number of join results due to the increased overlapping among tuples. The last property, record size, will not change the number of results but will affect the amount of memory required to process each join.

**Local Speed-Up on Synthetic Data.** Figure 3.19 shows the performance of the algorithms on a single node, using an overlapping interval join query over a synthetic dataset with 10,000 records evenly distributed over a time range of 1,000 units ($\lambda = 10$). Each interval has a duration $d$ equal to 10 and the tuple the size is 74 bytes. The overlapping join query is a self-join, so the same dataset is used on both sides of the join. All algorithms show good speed-up until eight partitions, at which point the processing node uses hyper-threading since it only has four cores.

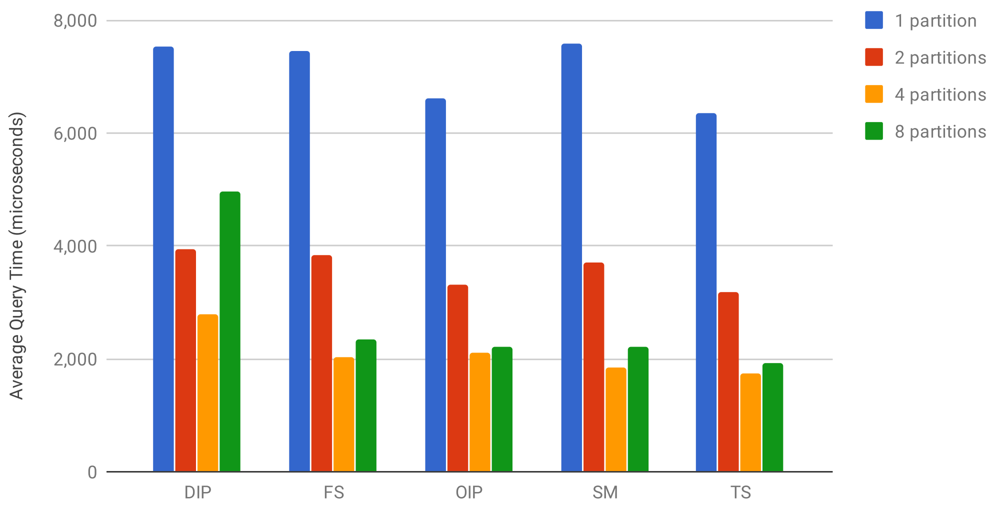


Figure 3.19: Single node speed-up performance for the baseline overlapping interval join query on synthetic data ($\lambda = 10$ and $d = 10$).

In Figure 3.20, the dataset cardinality has been increased ten times, while the time range and other properties have not been changed. As a result, the dataset has 100,000 records, a time range of 1,000, a density of $\lambda = 100$, and duration of $d = 10$. There are ten times as many in tuples in the dataset, and each such tuple will match (self-join) with tuples that have been increased by ten times. This results in 100 times more comparisons; as it can be seen in the figure the query time has increase similarly, when compared with Figure 3.19. Again each algorithm shows good speed-up.

In Figure 3.21, the dataset duration has been increased ten times. The self-joined dataset has now 10,000 records, a time range of 1,000, a density of $\lambda = 10$, and duration of $d = 100$. As a result, each tuple will now match with ten times as many tuples. The number of comparisons has increased by 10 times (when compared with Figure 3.19) due

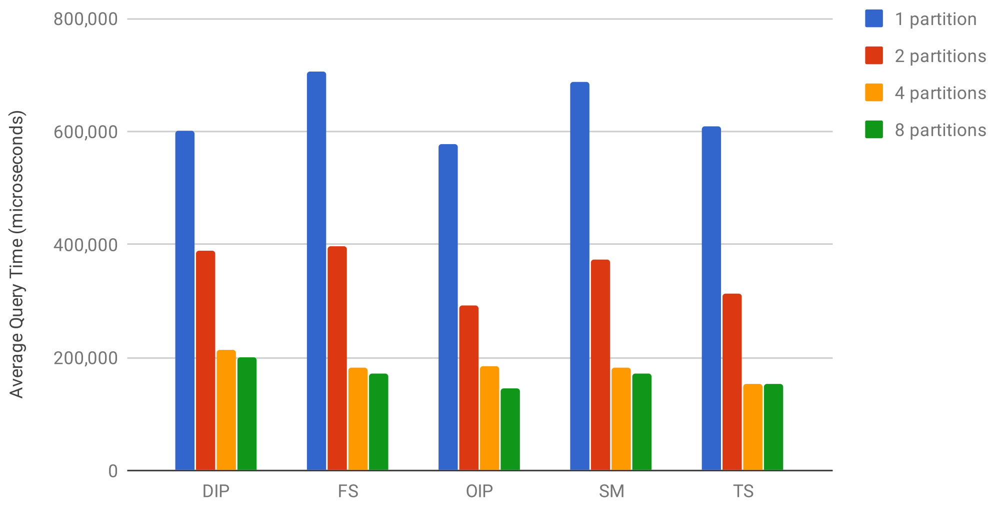


Figure 3.20: Single node speed-up performance for an overlapping interval join on synthetic dataset ($\lambda = 100$, and $d = 10$).

to the increased overlapping tuples while keeping a fixed cardinality of 10,000 tuples. The DIP results for this test are a special case. For each thread DIP creates 1,000 algorithm partitions; each of these algorithm partitions requires one frame (page) in memory. Since the available join memory has only 256 frames, the algorithm spills to disk. This happens for every thread. Except for DIP that is affected by spilling, the other algorithms show good speed-up performance under this scenario, too. We examine DIP's spilling performance further, in the scale-up experiments of this section.

Figure 3.22 shows how large records affect the performance in the local speed-up experiments. The same interval characteristics as in Figure 3.19 are used but here the record size has increased to 2,296 bytes. The key difference is the amount of data that must be pushed through the join operator has increased thus resulting in higher query times as compared with Figure 3.19. The non-partitioning algorithms (FS, SM and TS) show good speed-up. The partitioning algorithms have an issue with the amount of memory needed to

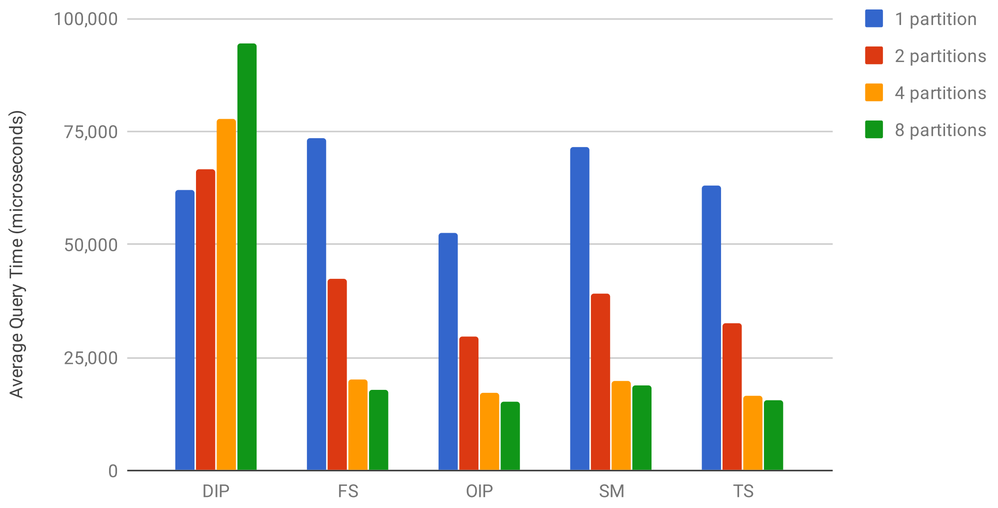


Figure 3.21: Single node speed-up performance for an overlapping interval join on synthetic dataset ($\lambda = 10$ and $d = 100$).

perform efficiently. DIP spills again to disk, but for a different reason. DIP creates the same number of algorithm partitions per thread as in Figure 3.19, thus there are enough frames for all algorithm partitions in memory; however, these frames spill because the records are much larger. Since there are enough memory frames the algorithm has fewer passes over the data than the previous spilling case. Overall, it shows good speed-up (affected though by disk contention since there is a single disk resource). OIP also spills for these experiments and the total I/O does not change as more threads are added. Since the cluster configuration uses only a single disk, the I/O bound OIP query time remains the same as threads are added.

**Real Datasets.** We used two datasets, infectious and TPC-H, to show how the algorithms perform on more realistic data. The infectious dataset [32] provides a real dataset with many short intervals with a low number of overlaps. The infectious dataset comes from an experiment tracking people's contact in an office building hallway. The researchers were

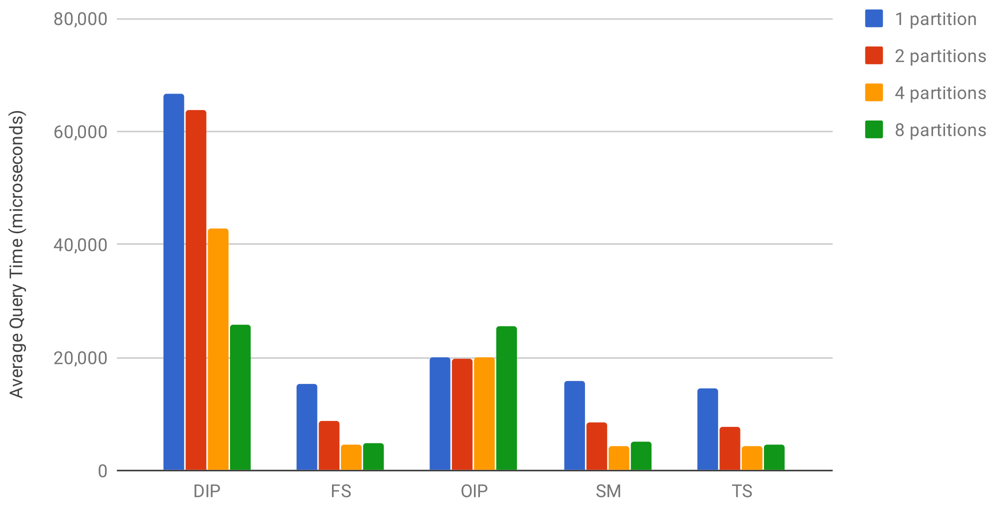


Figure 3.22: Single node speed-up performance for an overlapping interval join on synthetic dataset large records (2,296 bytes) ($\lambda = 10$ and $d = 10$).

looking into how epidemics spread in populations. The dataset holds pairs of people's ids and the starting time point of their contact. A contact corresponds to a 20-second interval where the two objects were both in the hallway (if a contact lasted for more than 20 seconds, a new record would appear). The dataset has 400 thousand records of contacts and at most 50 active contacts at each time instant. The dataset takes about 80 MB of disk space. The performance of the algorithms for the infectious dataset appears in Figure 3.23. All algorithms exhibit good speed-up performance on this dataset. Since the OIP algorithm was designed for joins that contain some long intervals, OIP had the worse query times on this dataset which has only small intervals.

The next dataset, TPC-H, is from a decision support benchmark. The benchmark includes a data generator that builds a dataset based on relevant industry data. One of the tables holds shipping times for ordered items. Each item entry has the dates when it was shipped and received, which we used to create an interval representing the time period

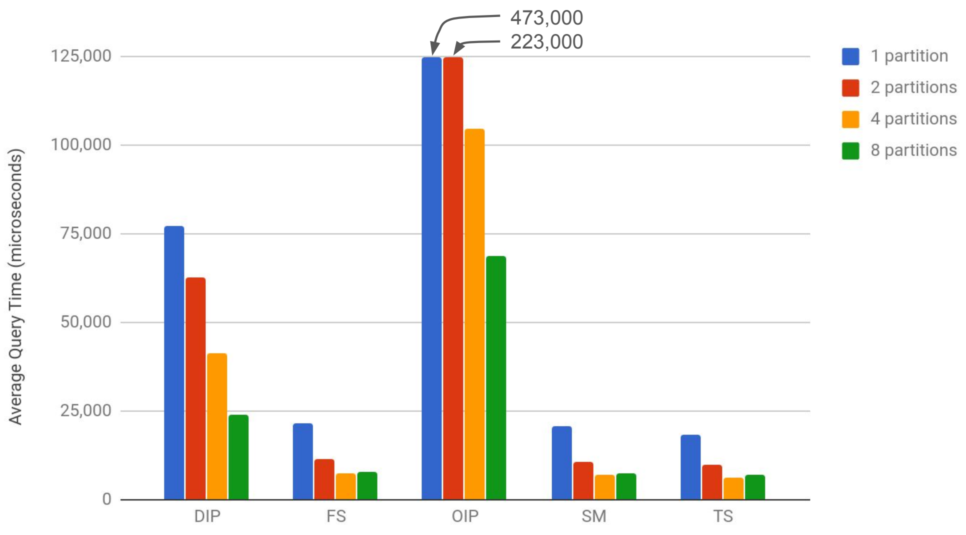


Figure 3.23: Overlapping join run over the Infectious dataset on a single node.

during which an item was in transit during shipping. In this example, 60 thousand records are self-joined in an overlapping query to create a result set of 44 million pairs of items in transit at the same time. Figure 3.24 shows the speed-up results; again all algorithms showed good speed-up performance; now OIP had overall faster query times.

**Multi-Node Speed-Up.** The local speed-up tests show that there is a benefit from adding partitions and threads up to the number of cores (4) of the local node. The next set of speed-up experiments focus on adding nodes to the cluster ("scaling out") while the number of partitions per node has been fixed to four. The first multi-node experiment has a synthetic dataset of 100,000 records, a time range of 10,000, a density $\lambda = 10$, and a duration $d = 10$. This baseline dataset setting is followed by changes to the cardinality, duration, and record size, similar to the local speed-up tests.

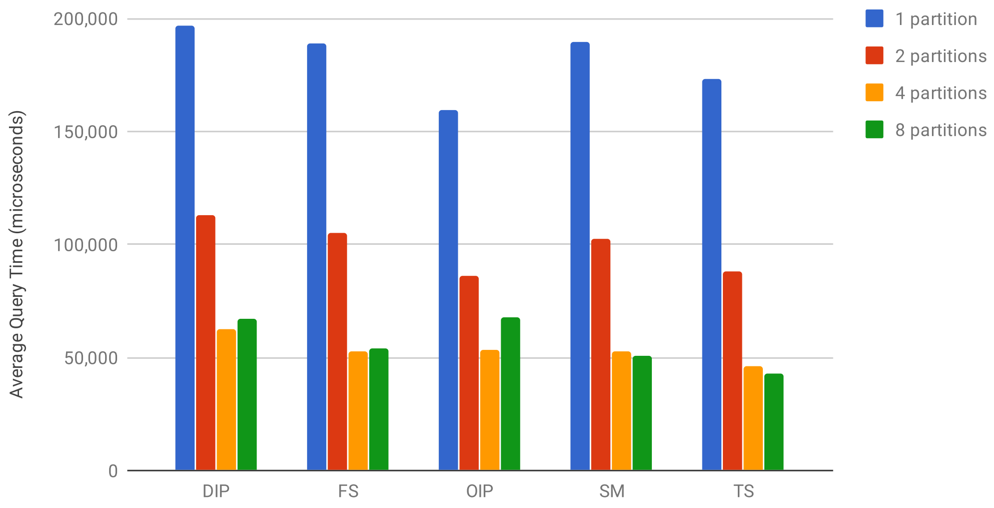


Figure 3.24: Overlapping join run over a TPC-H dataset on a single node.

Figure 3.25 shows that query execution times improve as the number of nodes is increased. However, the performance gain is slightly less than when adding threads in a single node because the network traffic from global partitioning starts to impact performance as the cluster's aggregate processing power increases.

Figures 3.26, 3.27, and 3.28 examine the cluster speed-up performance when changing the cardinality, duration and record size respectively. Overall, the algorithms show good speed-up performance for the cluster experiments, except for one DIP experiment. DIP in Figure 3.27 spills due to the large number algorithm partitions created to process the join; the same issue occurred for DIP in Figure 3.21. Also note that in the large record experiment in Figure 3.28, the partitioning algorithms (DIP and OIP) spill to disk, which causes slower performance in comparison to the other algorithms.

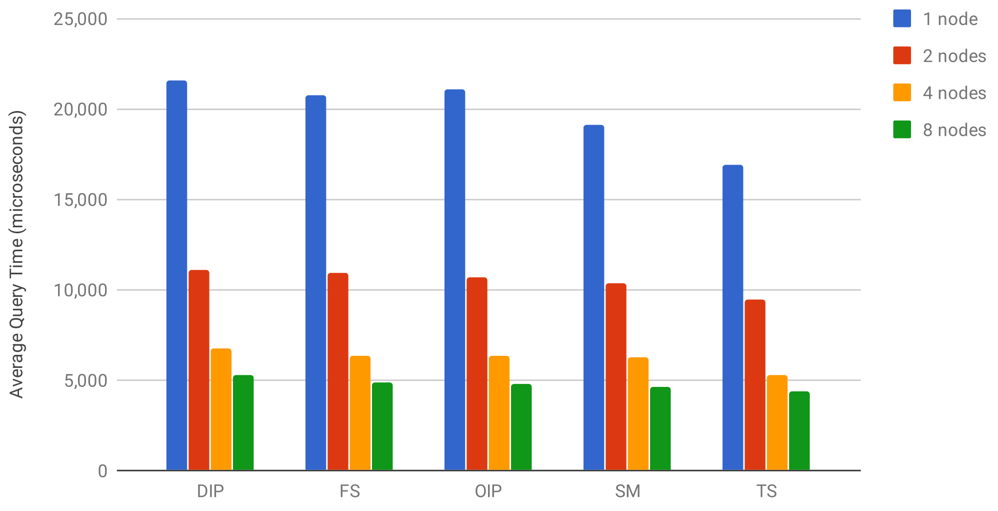


Figure 3.25: Speed-up performance for an overlapping interval join on a cluster with synthetic data ($\lambda = 10$ and $d = 10$).

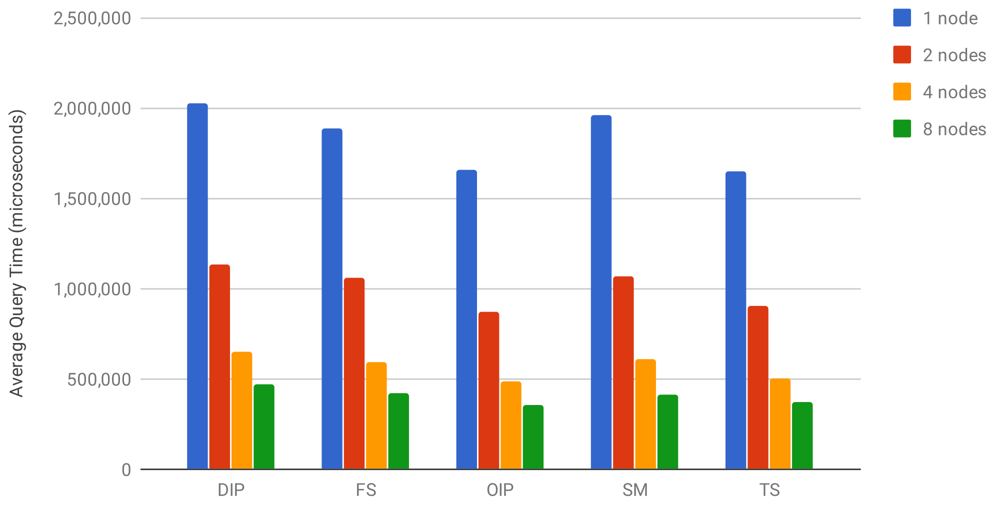


Figure 3.26: Speed-up performance for an overlapping interval join on a cluster with synthetic data ($\lambda = 100$ and $d = 10$).

### 3.4.2 Scale-Up Non-Spilling Experiments

The scaling experiments demonstrate how the system handles large data by keeping the data size the same per partition as the number of partitions is increased. In perfect

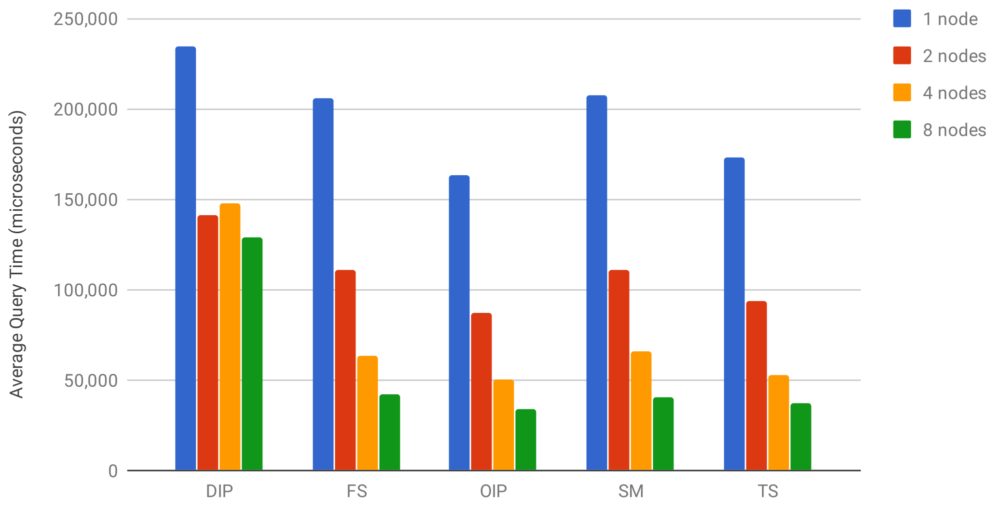


Figure 3.27: Speed-up performance for an overlapping interval join on a cluster with synthetic data ($\lambda = 10$ and $d = 100$).

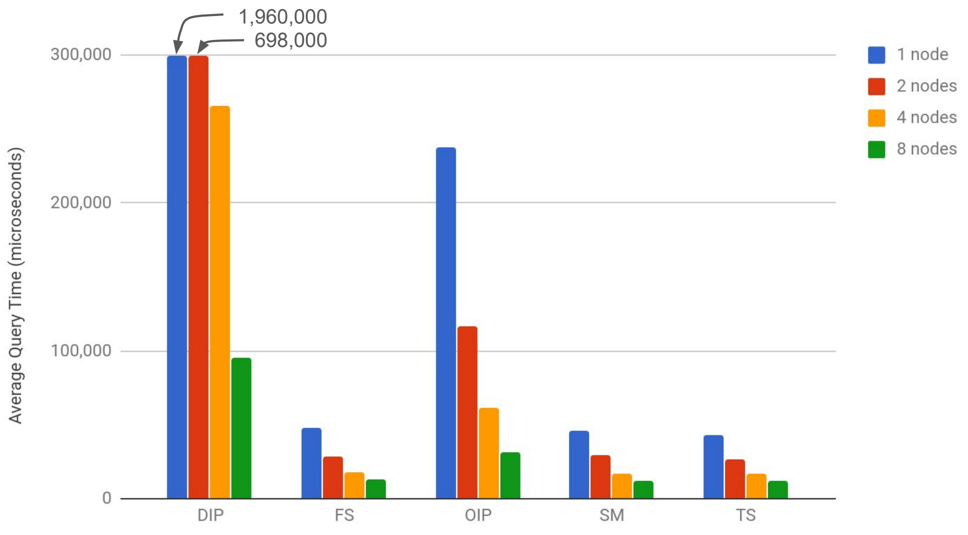


Figure 3.28: Speed-up performance for an overlapping interval join on a cluster with synthetic data using large records (2,296 bytes) ($\lambda = 10$ and $d = 10$).

scaling, as the dataset increases in size and resources are added, the query time would not change. The scale-up experiments were run for both single node with multiple partitions and for multiple nodes with a fixed number of partitions per node. Each partition in the dataset has 10,000 records, a time range of 1,000, a density of $\lambda = 10$, and a duration of $d = 10$. Figure 3.29 shows how the algorithms' scale-up on a single node from one to eight partitions. The scaling stops with four partitions since the node has four dedicated cores and hyper threads to eight cores.

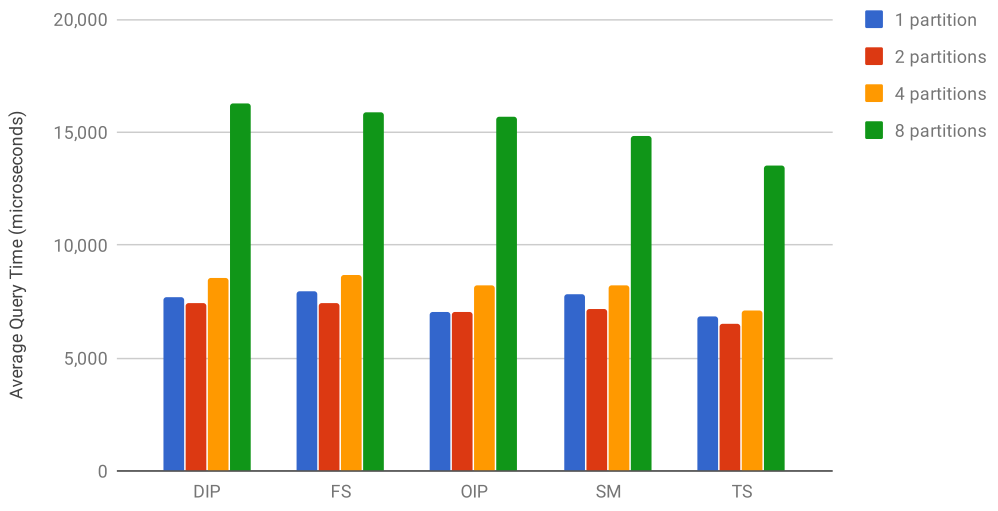


Figure 3.29: Scale-up on a single node which increase the number of partitions (1, 2, 4, and 8) where each partition has the same size of data.

The second scaling experiment, Figure 3.30, considers scale-up over multiple nodes. Here we show: one node (four partitions), two nodes (eight partitions), four nodes (16 partitions), and eight nodes (32 partitions). The queries on two or more nodes include network data transfers between nodes to arrange the data for local join processing. Going to two, four and eight nodes leads to additional network traffic, which slows the query down; otherwise, the algorithms handle scaling to many nodes fairly well.

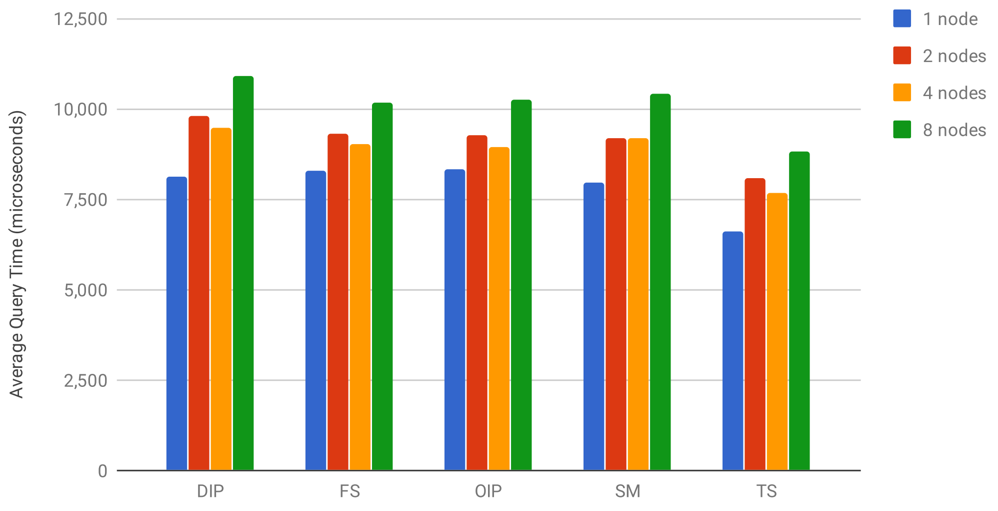


Figure 3.30: Scale-up for 1, 2, 4, and 8 nodes where each node holds four partitions each with the same size of data.

### 3.4.3 Scale-Up Spilling Experiments

Last but not least, the spilling experiments look at how the algorithms perform with limited memory. When dealing with limited memory, the size of tuples that flow through the query plan significantly affects the overall performance. If the "data does not stay with the record", the join process would create a list of all joined tuple pairs then another process must scan the datasets and create the new joined tuple for the result. Creating the result in a separate process would require another scan of one dataset and many random accesses to match the paired tuples with the other dataset. To understand the performance impact of larger tuples, we looked at various tuples sizes in the query pipeline. Table 3.2 shows the different tuples sizes tested in the query plan. When the query plan has both the primary key and the interval data, the input tuple is 74 bytes (from both relations), and when joined with the other side, the output tuple size is 148

| **Data** | **Padding** | **Input Tuple Size** | **Output Tuple Size** |
|---|---:|---:|---:|
| Id and Interval | 0 | 74 | 148 |
| Full Record | 222 | 296 | 592 |
| Full Record | 2,222 | 2,296 | 4,592 |

Table 3.2: Record size experiments and the effect on the input and output tuple size.

| **Join Memory** | **74 B Tuple** | **296 B Tuple** | **2296 B Tuple** |
|---:|---:|---:|---:|
| 2 MB | 28,160 | 6,912 | 768 |
| 32 MB | 453,376 | 113,152 | 14,592 |

Table 3.3: Number of input tuples needed to fill memory and force the algorithm to spill for various input tuple sizes and join memory sizes.

bytes. We tested increasing the record size by adding 222 bytes (a medium sized record) and 2,222 bytes (a large record).

Figure 3.31 shows how these record sizes affect performance. Carrying the larger tuples adds overhead in the pipeline due to the increased coping for larger tuples. The memory has been limited to 2MB for the join operator to ensure that each algorithm will make use of disk I/O to process the join query for the large record case (2,222 bytes). The increased tuple size leads to a sharp increase in query time.

The spilling scale-up experiments were configured so that each interval join algorithm will spill under these memory constraints. Each partition in the dataset has 100,000 records, a time range of 10,000, a density of $\lambda = 10$, a duration of $d = 100$, and a record size 2,222 bytes. The local scale-up experiments consider only tests with one and two partitions

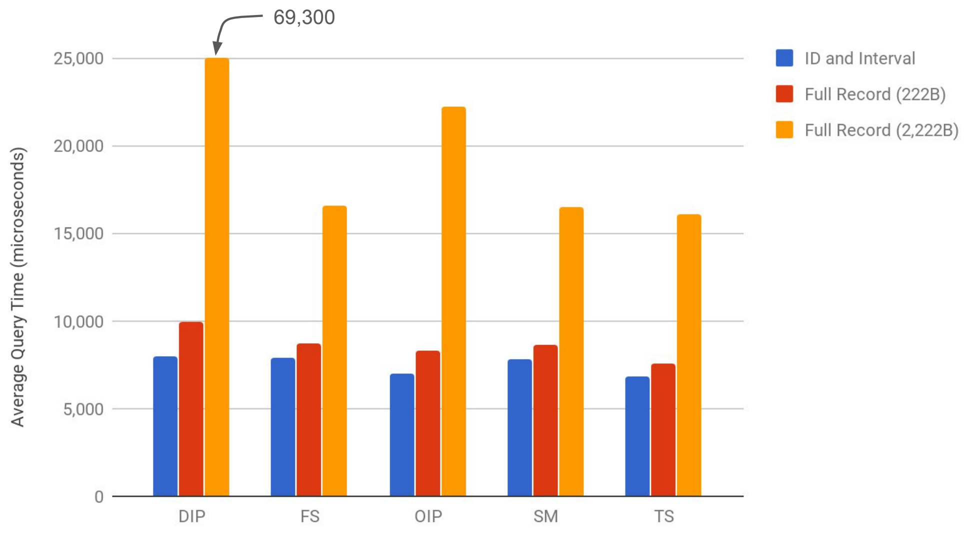


Figure 3.31: Performance test for increasing tuple size.

since the node has two disks. Note that the database logging happens on the same disks storing the data. The results of the single node scale-up experiments are in Figure 3.32. Each algorithm shows good scale-up behavior except DIP. We will discuss DIP scale-up performance later in this section.

The multi-node scale-up spilling experiments use one partition per node. A new (equal sized) partition is added with the addition of each node to the cluster. Note that data is written to one disk while the database logging is kept on the second disk, similar to the prior speed-up experiments. With the exception of DIP, the algorithms show good scale-up behavior from one node to eight nodes.

In the last two scale-up experiments, we noticed that DIP did not scale well, so we examined its behavior further. In particular, we examined how the number of partition files affects its performance. We performed a multi node scale-up experiment (with 1 to

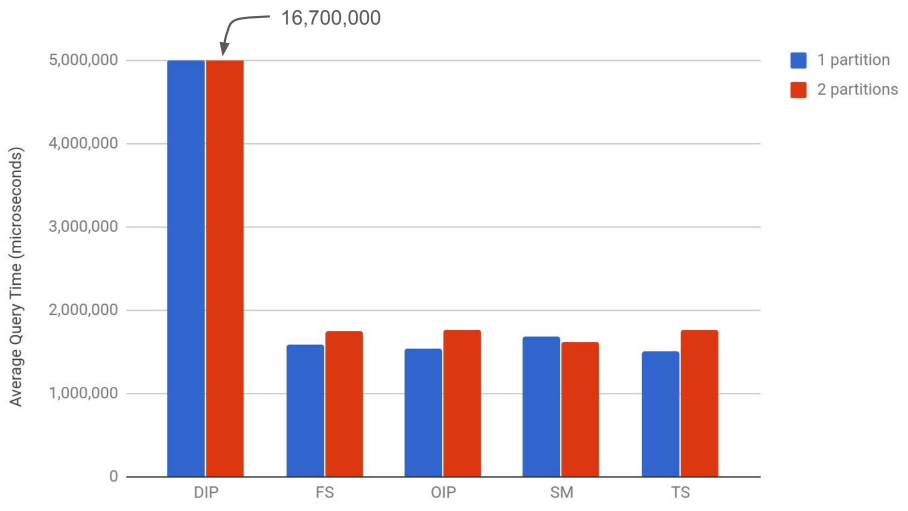


Figure 3.32: Single node scale-up spilling experiments with one and two partitions using large records and synthetic data ($\lambda = 10$ and $d = 100$).

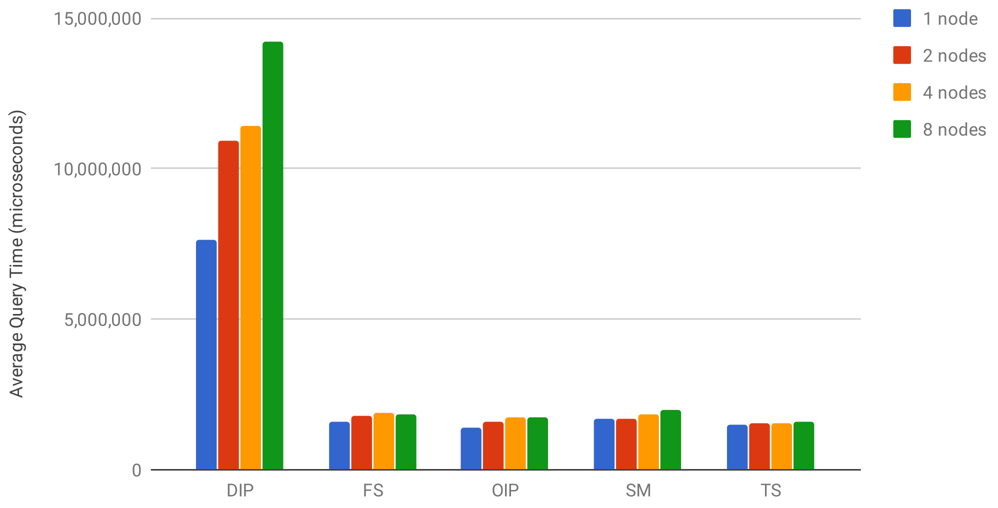


Figure 3.33: Multi-node scale-up spilling experiments on 1, 2, 4, and 8 nodes (one partition per node) using large records and synthetic data ($\lambda = 10$ and $d = 100$).

8 nodes), where each partition in the dataset has 10,000 records, a time range of 1,000, a density of $\lambda = 10$, a record size of 2,222 bytes and a varying duration. The duration was varied as follows $d = 10$, $d = 20$, $d = 40$, $d = 70$, $d = 100$, $d = 120$. As duration increases,

the number of algorithm partition files increases (since it is equal to the number of active tuples, namely: $\lambda d$). Figure 3.34 shows the results of this experiment. DIP exhibits good scale-up behavior up to $d = 20$ (i.e. 200 partition files) but its performance deteriorates as we reach $d = 120$ (1,200 partition files). One could consolidate all partition files into one, after the partition ends using one more pass (and before the join starts). This will enable the join to avoid various random I/Os, but requires elaborate partition markers in the file. While this is a promising idea, we leave it open for future research.

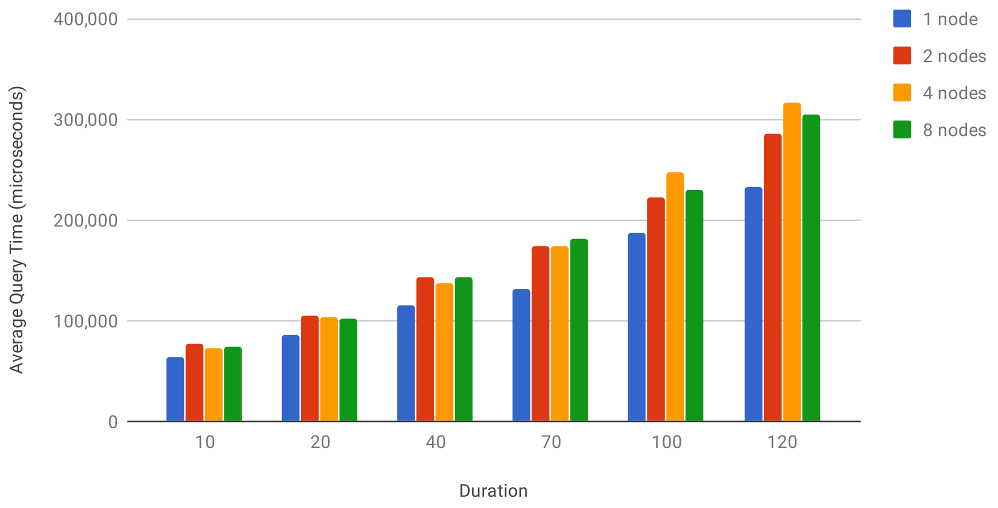


Figure 3.34: DIP multi-node scale-up spilling experiments on 1, 2, 4, and 8 nodes (one partition per node) using large records and synthetic data with $\lambda = 10$ while varying the duration $d$.

## 3.5 Conclusions

We have implemented five overlapping interval join algorithms in a real database system, Apache AsterixDB. The implementation required adding the *Split* global partitioner, adding a dynamic memory manager, creating a stream join operator, and updating the optimizer to recognize interval joins and pick a physical interval join operator. All five algorithms can now support spilling when memory is limited, which increases the opportunities to use these algorithms against big data. The algorithms' modifications presented allow the interval join algorithms to scale out gracefully across nodes and to work with limited memory.

Chapter 4

# Cost Models for Overlapping Interval Joins

## 4.1 Introduction

Each of the presented interval joined algorithms has a different approach to processing a join and method of utilizing the available memory budget. The CPU cost is based on the number of comparisons used to process the join. The I/O cost is based on the number of frames accessed from disk during the join. Further, the I/O cost model includes a method to determine if the algorithm is expected to spill. Due to the large difference in execution time between spilling and not spilling queries, this method can be used to determine if an algorithm spills (and thus select instead an appropriate algorithm to execute an interval join query without spilling, if possible). Finally, we note that these models have also been used to confirm the implementation of each algorithm.

To build the cost model, we considered a few key properties of the intervals associated with the dataset tuples: start of time range, end of time range, number of tuples, and average interval duration. To reduce the model complexity (and achieve closed formulas) we made two simplifying assumptions. First we assume that each tuple's interval has the same duration. Second that the interval start times are uniformly distributed over the time range. Then the arrival rate $\lambda_R$ at a given time instant (i.e. how many intervals start at that time instant) can be calculated by dividing the number of tuples $T_R$ in dataset $R$ by the dataset's time range: $\lambda_R = \frac{T_R}{r_{R.end} - r_{R.start}}$. Finally we note that the I/O cost model does not include the I/O for reading input or writing the result since this is the same for all algorithms. In Section 4.4.6, we discuss how the basic cost model can be expanded to describe a dataset that contains various classes of intervals (i.e., when the dataset contains a mixture of interval classes, where a class has its own arrival rate and average interval duration). Table 4.1 summarizes the notation used by our cost models.

An important part in understanding the cost models and spilling, is how the five algorithms use main memory; this is discussed in Section 4.2. Another important quantity is the size of the join result; Section 4.3 describes the join size estimation. The CPU and I/O cost models are presented for each of the five interval join algorithms in Section 4.4. Section 4.5, we presents the accuracy of the cost models while conclusions appear in Section 4.7.

| **Label** | **Description** |
|---|---|
| $R$ | Dataset R |
| $F_R$ | Size of Dataset R in frames |
| $T_R$ | Size of Dataset R in tuples |
| $t_R$ | Single tuple from Dataset R |
| $r_R$, $r_{R.start}$, $r_{R.end}$ | Dataset R's time range, start and end |
| $\lambda_R$ | Arrival rate in dataset R ($\lambda_R = \frac{T_R}{r_{R.end} - r_{R.start}}$) |
| $d_R$ | Average interval duration in dataset R |
| $L_R$ | Active tuples in R ($L_R = \lambda_R d_R$) |
| $b_R$ | Dataset R's Tuple size in bytes |
| $F_M$ | Memory size in frames |
| $T_M$ | Memory size in tuples ($T_M = \lfloor \frac{f}{b} \rfloor F_M$) |
| $f$ | Frame size in bytes |
| $T_f$ | Frame size in tuples ($T_f = \lfloor \frac{f}{b} \rfloor$) |

Table 4.1: Cost model notation (shown for Dataset $R$).

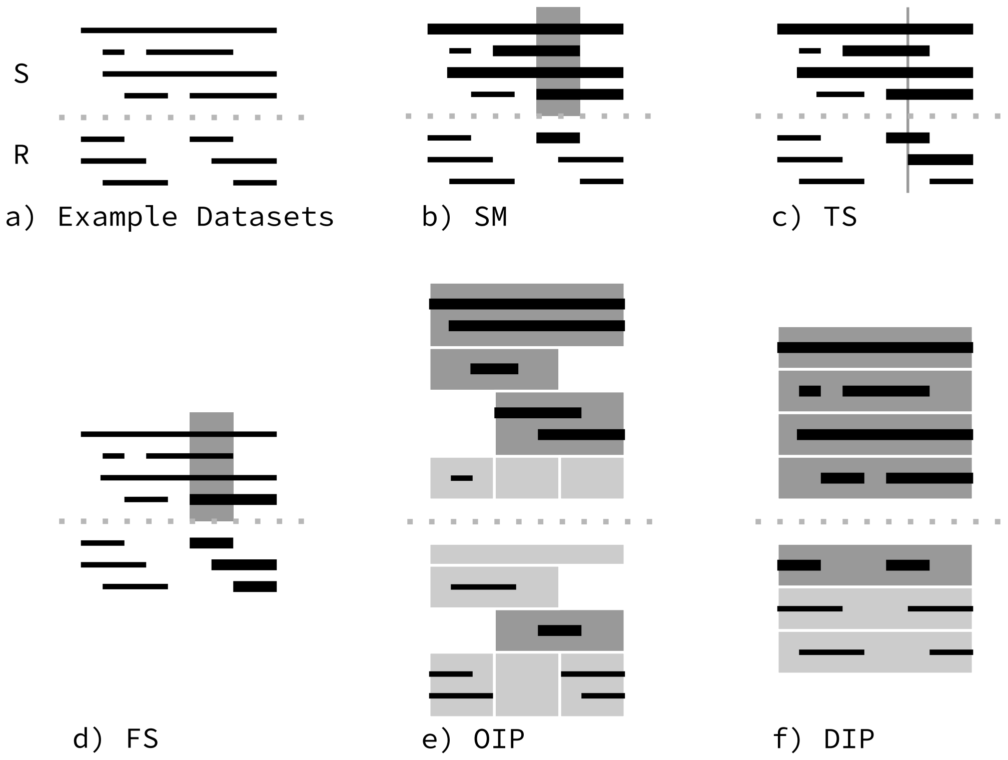


Figure 4.1: Main memory utilization for the interval join algorithms.

## 4.2 Main Memory Utilization

Figure 4.1 visualizes how main memory is used by each algorithm. It displays two datasets separated by a dotted line with $R$ on the bottom and $S$ on the top. The tuple intervals are depicted as line segments where time is increasing from left to right.

First, consider the non-partitioning algorithms. The SM algorithm in Fig4.1b. The first algorithm displayed is SM, shown in Figure4.1b. This algorithm keeps in memory one tuple ($t_R$) from $R$ and all the overlapping tuples with $t_R$ from $S$. The bold segments highlight intervals in memory and the overlapping range of $t_R$ is shown by a vertical gray region. The TS algorithm (Fig4.1c) keeps in memory all intervals that include a specific point in time; this time instant is depicted as a vertical line while the bold segments are those kept in memory. Note that TS must maintain active tuples from both $R$ and $S$ in memory. The FS algorithm manages memory similarly to SM, but applies the idea to both datasets. Figure 4.1d shows the case when FS has selected a tuple ($t_R$) in $R$ and loaded into memory any $S$ tuples (shown in the vertical grey region) that start during $t_R$. Note that $R$ already has tuples in memory that started during some previously selected $t_S$ tuple.

For the two partitioning algorithms we use horizontal grey boxes to illustrate the partitions created from partitioning each dataset. The OIP algorithm (Figure 4.1e) is using three slots and the intervals have been separated into partitions based on which slots they start and end. In this figure, we assume that partition (2,3) from dataset $R$ (shown as the darker grey box; it contains intervals that started in the 2nd slot and end in the 3rd slot) has been selected and its overlapping partitions from $S$ are loaded into memory (shown as dark grey boxes at the top). Only those partitions that contain intervals are loaded. The bold segments represent the intervals from those partitions that are in memory. The DIP algorithm creates partitions with non-overlapping intervals. Figure 4.1f displays the resulting disjoint partitions. The first disjoint partition from $R$ is loaded into memory and all disjoint partitions from $S$ will be in memory to process the join.

## 4.3 Join Size Estimation

For join size estimation we assume that datasets $R$ and $S$ cover the same time range (since disjoint time ranges will not produce results). Let's first assume that both $R$ and $S$ datasets fit in a single memory partition and include a tuple interval $t_R$. Clearly, $t_R$

will overlap with tuple intervals from $S$ that start during $t_R$'s duration ($d_R$). It will also overlap with $t_S$ tuples that have started at time instants that are at most $d_S - 1$ before the start of $t_R$. Thus the number of $S$ tuples that will match $t_R$ are $\lambda_S * (d_R + d_S - 1)$. As there are $T_R$ tuples in $R$, Equation 4.1 provides the number of overlapping interval results from joining dataset $R$ and $S$.

$$JE(R,S) = T_R * \lambda_S * (d_R + d_S - 1) \tag{4.1}$$

This formula overestimates the join result in two ways. First, it assumes that for each tuple $t_R$ there are $d_S - 1$ time instants before its start time where tuples from $S$ can start. However, $t_R$ tuples that start in the first $d_S - 1$ time instants in the time range will not have all those $S$ matching tuples. The formula $JE_{before}(R,S)$ (4.2) calculates this overcounting. A similar case involves the matchings of $t_R$ tuples that extend beyond the end of the time range, calculated by $JE_{after}(R,S)$.

$$\begin{aligned} JE_{before}(R,S) =& \lambda_R * \lambda_S * \sum_{i=1}^{d_R-1} i = \lambda_R * \lambda_S * \frac{(d_R-1)*d_R}{2} \\ JE_{after}(R,S) =& \lambda_R * \lambda_S * \sum_{i=1}^{d_S-1} i = \lambda_R * \lambda_S * \frac{(d_S-1)*d_S}{2} \end{aligned} \tag{4.2}$$

As a result, the join result estimate for the single node case is given by:

$$JE_{single}(R,S) = JE(R,S) - JE_{before}(R,S) - JE_{after}(R,S) \tag{4.3}$$

When the joined relations cannot fit in one partition, a shared-nothing system (like AsterixDB) can the *SPLIT* partitioning strategy to divide each dataset into many partitions. Each partition is responsible for a non-overlapping portion of the dataset's time range. As a result, there is an initial partition representing the start of the time range, followed by zero or more middle partitions and the last partition which is responsible for the end of the time range. Respective partitions in $R$ and $S$ (say $R_k, S_k$) share the same portion of the time range. When estimating the number of join results in a partition one must consider if this partition is *first*, *middle* or *last* in the time range so as to eliminate result overestimation. The following formulas are used to calculate the join size estimation for those individual cases.

$$\begin{aligned} JE_{first}(R,S) =& JE(R,S) - JE_{before}(R,S) \\ JE_{middle}(R,S) =& JE(R,S) \\ JE_{last}(R,S) =& JE(R,S) - JE_{after}(R,S) \end{aligned} \tag{4.4}$$

The join result size estimation is important as it enables a cost-based query model to calculate the number of tuples expected as output from the interval join operator. To explore the accuracy of the above formulas, we ran experiments over a single partition, while varying the duration and arrival rate and calculated the error between the actual number of results and our estimates ($\%error = |(actual - estimated)|/estimated * 100\%$). Table 4.2 shows samples from a self join where the dataset's interval duration $d$ changes from 1 to 100 time instants while the time range is fixed to 1,000 instants and the arrival rate is 10. Table 4.3 considers a self join where we varied the interval arrival rate ($\lambda$) while the time range is fixed to 1,000 instants and the duration is 10. Since the time range does not change, the arrival rate changes the total number of intervals (or cardinality) in the dataset. However the duration in this case does not change, hence the overlap is not affected and

| **Duration** | **Join Size Estimation Error** |
|---|---|
| $d = 1$ | 0.00% |
| $d = 5$ | 0.02% |
| $d = 10$ | 0.05% |
| $d = 50$ | 0.25% |
| $d = 100$ | 0.50% |

Table 4.2: The join size estimation cost model error for various durations where the time range is fixed at 1,000 instances and $\lambda = 10$.

thus the error remains the same. Overall, the tables show that the join estimation formulas are fairly accurate.

## 4.4 Cost Models

We next present CPU and I/O cost models for these five algorithms: Sort-Merge, Time-Sweep, Forward-Scan, Overlapping Interval Partition Join, and Disjoint Interval Partition Join. One assumption shared by all algorithms is that the input relations are ordered by their intervals' start time (followed by their end time). Hence, the models below do not consider the prior sorting phase as it is equivalent for all algorithms.

### 4.4.1 Sort-Merge Interval Join Model

The sort-merge interval join is similar to a typical sort-merge join which contains many duplicates (where those duplicates behave as an overlapping interval). The cost model

| **Lambda** | **Join Size Estimation Error** |
|---|---|
| $\lambda = 1$ | 0.05% |
| $\lambda = 5$ | 0.05% |
| $\lambda = 10$ | 0.05% |
| $\lambda = 50$ | 0.05% |
| $\lambda = 100$ | 0.05% |

Table 4.3: The join size estimation cost model error for various lambdas where the time range is fixed at 1,000 instances and $d = 10$.

focuses on the merge phase where tuples from the two input streams are scanned to build the join.

The join memory becomes full when a tuple $t_R$ from $R$ overlaps with more tuples in $S$ than can fit into memory (see Figure 4.2). The overlapping range of $t_R$ with tuples in $S$ is $d_R + d_S - 1$. Multiplying that range by the arrival rate in $S$ ($\lambda_S$) gives the number of $S$ tuples that will overlap $t_R$ (using Little's formula [39]). To avoid spilling these tuples must be in memory at the same time. The number of frames needed to store these tuples is computed by dividing the number of tuples by the tuples per frame, or $T_{f.S} = \lfloor \frac{f}{b_S} \rfloor$, which is the number of tuples of size bytes $b_S$ from $S$ that can fit into a frame of size $f$ in bytes. The join algorithm also uses four frames that reduce the total join memory available: a frame for each of the two input streams $R$ and $S$, one frame for the output result and one for managing the spilled dataset. The remaining memory is used by sort-merge interval join to store $S$ tuples for join processing. Thus, the join algorithm exceeds available memory ($F_M$) and spills when the following inequality is satisfied:

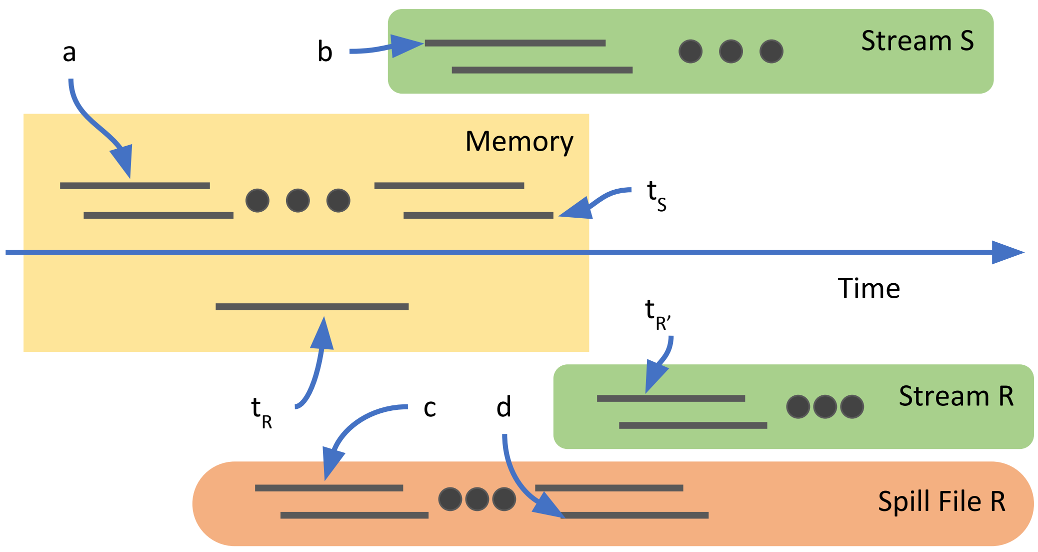


Figure 4.2: Important tuples during the spilling process for sort-merge interval join. The colored blocks represent intervals in memory, the input streams for $R$ and $S$ and the spill file for $R$. Tuple $a$ corresponds to the first $S$ tuple currently in memory, tuple $t_S$ is the last $S$ tuple in memory and tuple $b$ is the next tuple from the $S$ stream. Similarly, tuple $t_R$ corresponds to the active $R$ tuple in memory, tuple $c$ is the first tuple in the spill file of $R$, tuple $d$ corresponds the last tuple in that spill file, and tuple $t_{R'}$ is the next tuple from the $R$ stream.

$$\frac{\lambda_S * (d_R + d_S - 1)}{T_{f.S}} > F_M - 4 \tag{4.5}$$

The spill process starts with the first tuple, $t_R$, that overlaps more $S$ tuples than what can fit into memory. When this happens, the algorithm pauses reading tuples from $S$. The spill process focuses on freeing space by removing $S$ tuples from main memory. Before an $S$ tuple can be removed from main memory, the spill process must ensure the tuple has already been matched with all possible $R$ tuples (including future tuples from the stream). Thus the algorithm will proceed by reading tuples from $R$, until the matches for all $S$ tuples currently in memory have been found. At this point the entire memory resident set of $S$ tuple can be erased from main memory, since all match tuples from $R$ have been made with these $S$ tuples. However, these $R$ tuples may have matches with future $S$ tuples from the stream, hence they cannot be thrown away. This is why we spill (future) $R$ tuples to a temporary file to allow for these $R$ to match with future $S$ tuples.

The sort-merge interval join may have many spills. The size of these spills varies based on the relative properties of the two input streams. Note that while the $S$ and $R$ streams may begin at separate times, we start matching only when they overlap. For a given time instant, what is important is how many tuples from each stream are 'active' during (i.e. contain) this time instant. Based on our assumption of a steady arrival rate and a fixed tuple interval duration for each input stream, there are three phases: (1) a *warm-up* phase where the number of 'active' tuples keeps increasing, (2) followed by a *steady* state, where the number of active tuples remains constant, and (3) the *cool-down* phase where the number of 'active' tuples decreases, eventually returning to zero.

As a result, during the course of the sort-merge interval join, the size of the spills follows the same pattern, as seen in Figure 4.3. During the warm-up phase the spill starts from an initial size (depicted as 'a' in the figure) that incrementally increases (adding 'b' increments) until it reaches its steady-state size (depicted as 'c'). Then during the cool-down phase, the spill starts decreasing (by decrements 'b') until it reaches the last spill size depicted as 'd'.

To create the I/O cost model, we start by calculating the size of each of these spills. Next, we proceed with determining how many spills occur during each of these phases. Finally, using the size and number of spills, we calculate the total read and write I/O for the sort-merge interval join.

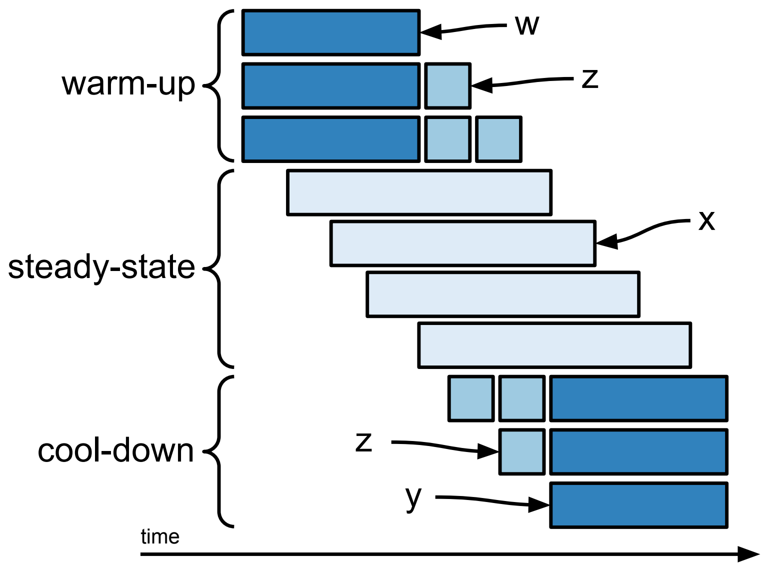


Figure 4.3: The $R$ spilling phases for interval merge join. The spill file denoted as $w$ represents the very first spill, while $z$ represents the spill increment, $x$ the spill file during the steady state, and $y$ the very last spill file.

**Size of the spill files.** We proceed with the calculation of the spill size for the 'steady' state, followed by the size of last spill, first spill, and the increment spill. For each spill process initiated, a spill file is created to store tuples from $R$ that match with $S$ tuples in memory. When considering the spill file at steady state (shown in Figure 4.3 as 'x' spill) the algorithm should find all matches for the current memory resident $S$ tuples. These matches are new tuples (after $t_R$) from the $R$ stream. (see Figure 4.2.) This process reads tuples from $R$ until we reach a tuple in $R$ (say $t_{R'}$) that starts after the furthest end point of any $S$ tuple currently in memory. This guarantees that the $S$ tuples currently in memory cannot overlap with any future tuple in the $R$ stream. Since the $R$ tuples between $t_R$ and $t_{R'}$ may also match future $S$ tuples, they need to be saved to a spill file and processed later. The time range represented by the start of $t_R$ until the start of tuple $t_{R'}$ is defined as $d_R + d_S - 1$, and explained next. (This formula assumes that tuples matching this time range exist in $R$; as with the join estimation case, there are two extreme cases, in the beginning and the end of the $R$ stream, that need special attention and will be discussed as part of the first and last spill).

The $t_R$'s duration, $d_R$, represents a time range for start points of all $S$ tuples in memory that have been added to memory due to overlapping with $t_R$. Consider an $S$ tuple ($t_S$) in memory which has the latest end time among those $S$ tuples currently in memory; this tuple could overlap with $R$ tuples for all time instances in its own duration $d_S$. Since this $S$ tuple's duration does not include its endpoint, the overall range is decreased by one.

As a result, the number of $R$ tuples that are in the range represented by $S$ tuples in memory is given by: $\lambda_R(d_R + d_S - 1)$. Let $T_{f.R} = \lfloor \frac{f}{b_R} \rfloor$ be the number of tuples of size $b_R$ bytes from $R$ that can fit into a frame of size $f$ in bytes. The number of I/O frames needed to write the spilled $R$ tuples in the spill file is thus:

$$C_{IO.steady.spill}(R, S) = min(F_R, \frac{\lambda_R}{T_{f.R}}(d_R + d_S - 1)) \tag{4.6}$$

With every spill that happens, the start time of tuple $t_S$ (Figure 4.2) approaches the start time of the final tuple in relation $R$. Consider the situation when the last spill occurs (depicted as 'y' spill in Figure 4.3). It may be that the tuple $t_S$ starts exactly at the start time of the final tuple in $R$. Because this is the final $R$ tuple, no other tuples can spill from $R$ during $d_S$ (the duration of $t_S$). In order to calculate $C_{IO.last.spill}(R, S)$, i.e., the size of the very last spill file, the temporal range becomes: $d_R + 1 - 1$, or simply $d_R$. Thus:

$$C_{IO.last.spill}(R, S) = \ min(F_R, \frac{\lambda_R}{T_{f.R}} d_R) \tag{4.7}$$

Similarly to the last spill file, the first $R$ spill file (depicted as ‘w’ in Figure 4.3) has a different range of overlapping tuples. As tuples from $S$ are loaded into memory, it may be that $t_S$ (the last $S$ tuple in memory) starts at the same time instance as tuple $t_R$ (the initial $R$ tuple). In this case, instead of loading all $S$ tuples for the temporal range of $d_R$, memory can be filled after loading only a single time instance from $S$. Thus the temporal range becomes $1 + d_S - 1$, or simply $d_S$. Hence $C_{IO.first.spill}(R, S)$, the size of the very first spill file is:

$$C_{IO.first.spill}(R, S) = \ min(F_R, \frac{\lambda_R}{T_{f.R}} d_S) \tag{4.8}$$

We next consider the spill increment size (represented by ‘z’ in Figure 4.3) that will bring the initial spill file to its steady state and then incrementally decrements it to the final spill file size. The reason the algorithm has spilled is because the overlapping tuples of $S$ could not fit into memory. Each time memory spills, the number of additional tuples written to the spill file is based on the additional interval represented in memory by $S$. Consider the actual number of $S$ tuple in memory, $T_{M.S}$, and the interval of time represented by these tuple’s start points, calculated by dividing $T_{M.S}$ by the number of $S$ tuples starting at a time instance, $\lambda_S$. What tuples from $R$ exist in the same the time interval as these $S$ tuples in memory? Multiplying by $\lambda_R$ gives the number of $R$ tuples in the time interval of memory. Consider if these tuples $R$ tuples were saved to a spill file, dividing the number of tuples by $T_{f.R}$ converts the number of tuples to frames of $R$ tuples. The resulting formula is used for number of new frames during the warm-up phase and the number of frames removed during the cool-down phase. The final formula for increment spill is:

$$C_{IO.spill.inc}(R,S) = \frac{\lambda_R}{T_{f.R}} \frac{T_{M.S}}{\lambda_S} \tag{4.9}$$

$C_{IO.spill.inc}(R,S)$ is the same for change in the spill size from the first spill size to the stead state. With each flush of $S$ from memory, the spill file will grow $C_{IO.spill.inc}(R,S)$ until it reaches the full size of $C_{IO.steady.spill}(R,S)$.

**Number of spill files.** The next step is to determine the number of spills for each of the three phases: warm-up, steady state, and cool-down. First consider $I_{total}(S)$, the total number of spills that occur during the join. To enter a spill process, memory will be full of $S$ tuples. After each spill the tuples from $S$ in memory are no longer needed for the join. Thus $I_{total}(S)$ is upper bounded by the ratio of the size of dataset $F_S$ and the size of memory $F_M$:

$$I_{total}(S) = \left\lfloor \frac{F_S}{F_M} \right\rfloor \tag{4.10}$$

We will proceed with computing the number of spills in the cool-down phase, followed by the warm-up and finally the steady state. The cool-down phase uses two types of spills: the last spill and the increment spill. Working backwards, during each new spill, the last spill will be augmented by the increment spill until it reaches the steady state size. A particular spill in this phase represents the last spill plus some number of increment spills to create the current spill size. If the final spill size is $C_{IO.last.spill}(R,S)$ and each preceding spill is increased by $C_{IO.spill.inc}(R, S)$ until the size reaches $C_{IO.stead.spill}(R, S)$, then the number of spills in the cool-down phase $I_{coolDown}(R, S)$ is given by:

$$I_{coolDown}(R, S) = \left\lfloor \frac{C_{IO.steady.spill}(R, S) - C_{IO.last.spill}(R, S)}{C_{IO.spill.inc}(R, S)} \right\rfloor \tag{4.11}$$

The same argument can be applied to the warm-up phase in relationship to the initial and steady state spills. If each spill file grows by $C_{IO.spill.inc}(R, S)$ starting from $C_{IO.first.spill}(R, S)$ until the size reaches the steady state of $C_{IO.steady.spill}(R, S)$, then the number of spills in the warm-up phase $I_{warmUp}(R, S)$, is given by:

$$I_{warmUp}(R, S) = \left\lfloor \frac{C_{IO.steady.spill}(R, S) - C_{IO.first.spill}(R, S)}{C_{IO.spill.inc}(R, S)} \right\rfloor \tag{4.12}$$

As a result, the number of spills at the steady state $I_{steady}(R, S)$ is given by:

$$I_{steady}(R, S) = I_{total}(S) - I_{warmUp}(R, S) - I_{coolDown}(R, S) \tag{4.13}$$

**Calculating the I/O Cost.** The I/O cost is determined by the number of frames written and read during the sort-merge interval join. First consider the number of frames written using Figure 4.3 as a guide. When the sort-merge interval join algorithm resumes after a spill, it first starts reading tuples from the spill before it continues processing relation $R$ (recall that it had paused at tuple $t_{R'}$). Nevertheless, another spill can happen while reading from the spilled $R$ tuples. One approach could be to write a new spill file and read from there when the sort-merge algorithm restarts, as shown by each row in Figure 4.3. However, we can save writes by appending to the existing spill file (with tuples from $R$ that were spilled and are after $t_{R'}$). This single spill file may be read multiple times (as new spills may occur) but such reads are easy to manage with appropriate pointers to its tuples (keeping track of where each spill effectively starts). By appending the spill file (and re-reading it) the total number of frames written to the spill file is upper-bounded by the size of $R$.

$$C_{IO.written}(R,S) = F_R \tag{4.14}$$

Now consider the number of frames read during the three spilling phases. The complete spill file will be read for each spill. The read I/O can be calculated using the formulas for the size of each spill and the number of spills. Thus, the read I/O during the steady state is given by:

$$C_{IO.steady}(R,S) = I_{steady}(R,S) * C_{IO.steady.spill}(R,S)) \tag{4.15}$$

The read I/O for the cool-down spill phase is given by the following summation:

$$C_{IO.coolDown}(R,S) = \sum_{i=0}^{I_{coolDown}(R,S)-1} (i * C_{IO.spill.inc}(R,S) + C_{IO.last.spill}(R,S)) \tag{4.16}$$

In this summation, the case $i = 0$ corresponds to reading the last spill file which has size $C_{IO.last.spill}(R,S)$. Then this spill file is incremented by $C_{IO.spill.inc}(R,S)$ to create the second to last spill file, etc. until $i = I_{coolDown}(R,S) - 1$.

In the warm-up spill phase we start from the first spill which has size $C_{IO.first.spill}(R,S)$. The second spill has size of $C_{IO.first.spill}(R,S) + C_{IO.spill.inc}(R,S)$ (as shown in Figure 4.3 by the top two spill rows). The next spill adds another $C_{IO.spill.inc}(R,S)$ frames to the spill file, etc. As a result, the read I/O for the warm-up phase is calculated by the following summation:

$$C_{IO.warmUp}(R,S) = \sum_{i=0}^{I_{warmUp}(R,S)-1} (i * C_{IO.spill.inc}(R,S) + C_{IO.first.spill}(R,S)) \tag{4.17}$$

The total I/O cost $C_{IO}(R,S)$ sums the frames written and read through the three spill phases:

$$C_{IO}(R,S) = C_{IO.written}(R,S) + C_{IO.steady}(R,S) + C_{IO.coolDown}(R,S) + C_{IO.warmUp}(R,S) \tag{4.18}$$

**Calculating the CPU Cost.** The CPU cost can be computed by the size of the result (which we already estimated as $JE(R,S)$ in Section 4.3) plus an estimate of the number of comparisons that did not produce any result. To estimate the non-result producing comparisons, we note that the algorithm keeps comparing $t_R$'s interval with matching $S$ tuples until it reaches the first $S$ tuple that does not overlap (at which point it proceeds to the next $t_R$ tuple). Thus, the total number of non-result producing comparisons is $T_R$, the number of tuples in the stream $R$. Again we need to consider for overestimation. Consider the last tuple in $S$; there may be various $R$ tuples that overlap the interval of the last tuple in $S$. These $R$ tuples do not produce any non-result comparison, simply because there is no more $S$ tuple. As there are $\lambda_R * (d_R - 1)$ such $R$ tuples, the total CPU cost is:

$$C_{CPU}(R,S) = JE(R,S) + T_R - \lambda_R * (d_R - 1) \tag{4.19}$$

Note that the SM algorithm has a special case for when the duration is one. In this case the algorithm does not need to make any unproductive comparisons.

### 4.4.2 Time-Sweep Interval Join Model

The time-sweep interval join (TS) algorithm processes the join by sweeping through time, instance by instance. In this case, the algorithm sweeps through the time instances stored in the tuples intervals. The tuples stored in memory are the ones currently active for the time instance the algorithm is currently processing. Using Little's formula [39] we can get the number of active tuples from each dataset, namely $L_R$ and $L_S$ (see Table 4.1).

The TS algorithm will spill if the active tuples from each dataset exceeds the available join memory. Since memory is referenced in frames, the active tuples must be converted into frames using the number of tuples in a frame for each dataset, $T_{f.R}$ and $T_{f.S}$ respectively. The join memory must also account for the five buffers (frames) TS uses for performing the join: a frame for each of the two input streams $R$ and $S$, one frame for the output result, and one frame for each of the spilled datasets for $R$ and $S$. Considering these buffers and Little's formula, the algorithm will spill when the following formula is true:

$$\frac{L_R}{T_{f.R}} + \frac{L_S}{T_{f.S}} > F_M - 5 \tag{4.20}$$

The spill process starts when the next tuple in time order can not be added to memory. The difference here is that this tuple may come from either dataset. If memory becomes full and the next time ordered tuple can not be added to memory, the TS algorithm pauses to free memory. In an effort to free the most memory, the dataset with the *most tuples in memory* is chosen for removal (after all its matches are found). Assume that $S$ has the most tuples in memory. To free these $S$ tuples, the tuples must be matched with upcoming $R$ tuples. The join process will continue to read $R$ tuples and match them with

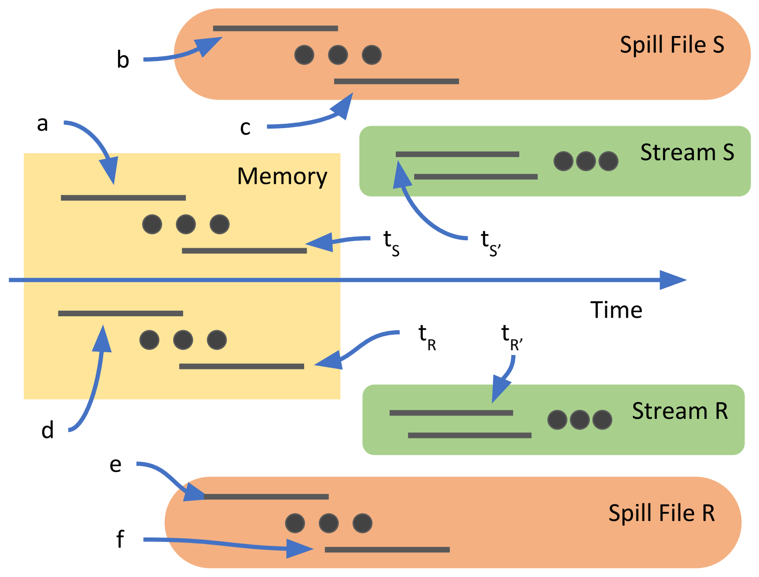


Figure 4.4: Important tuples during the spilling process for the time-sweep interval join. The colored blocks represent the intervals in memory, the input streams for $R$ and $S$ and the spill files for $R$ and $S$. Tuple $a$ corresponds to the first $S$ tuple currently in memory, tuple $t_S$ is the last $S$ tuple in memory, tuple $b$ is the first tuple in the spill file of $S$, tuple $c$ corresponds to the last tuple in that spill file, and tuple $T_{S'}$ is the next tuple from the $S$ stream. Similarly, tuple $d$ is the first $R$ tuple currently in memory, tuple $t_R$ corresponds to the last $R$ tuple in memory, tuple $e$ is the first tuple in the spill file of $R$, tuple $f$ corresponds the last tuple in that spill file, and tuple $t_{R'}$ is the next tuple from the $R$ stream.

memory until all in memory $S$ tuples have been matched. Since future $S$ tuples may match with these $R$ tuples, they are saved to a spill file to be read later. The same spill process is used for when $R$ has the most tuples in memory and, in that case, a spill file of $S$ would be created.

The join algorithm may spill many times while processing the join. The size of each spill file depends on the data input and which dataset is being spilled. Similar to SM, the TS algorithm has several spill phases. The TS algorithm loads tuples into memory from both datasets based on time order. Since all tuples are processed in time order from both datasets, the algorithm does not have a warm-up spill phase, only the steady-state

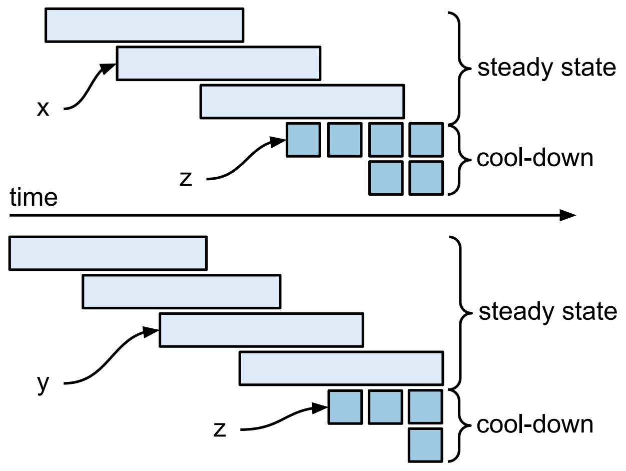


Figure 4.5: The $R$ and $S$ spilling phases for time-sweep interval join. The spill file denoted as $x$ represents the steady state spill file for relation $S$, while $y$ represents the spill increment, and $z$ the spill file during the steady state for relation $R$.

and cool-down phases as shown in Figure 4.5. The next spilled partition is determined by the dataset with the most tuples in memory. Figure 4.5 highlights an example set of spills while performing the TS join with each spill alternating between the two datasets and the two spill phases. To create the I/O cost model, we continue with the same steps from SM. First the size of each spill is calculated, then the number of spills is determined, which is used to create the complete I/O cost model for time-sweep interval join.

**Size of the spill files.** Assume again the case in steady state where the largest memory partition is for $S$. To free the memory currently occupied by $S$ tuples, will require matching those tuples with tuples from the stream of $R$. As these $R$ tuples will be needed for future matchings with $S$ tuples, they are saved in a spill file for $R$. The size of the spill file is determined by time range of tuple $t_S$ (the tuple from $S$ with the latest end time) currently in memory. While in the SM algorithm we consider tuples in $R$ that either start or end within the duration of $t_S$ ($d_S$), TS only needs to consider $R$ tuples that start during $d_S$. Thus the size of the spill file when $R$ is spilled, $C_{IO.steady.spill}(R,S)$, is given by:

$$C_{IO.steady.spill}(R,S) = min(F_R, \frac{\lambda_R}{T_{f.R}}(d_S - 1)) \tag{4.21}$$

If instead, $S$ was spilled, the spill file $C_{IO.steady.spill}(S,R)$, is given by:

$$C_{IO.steady.spill}(S,R) = min(F_S, \frac{\lambda_S}{T_{f.S}}(d_R - 1)) \tag{4.22}$$

Next we consider the cool-down phase, by looking at the last partition. Since TS progresses through each tuple instance by instance, the last spill file is based on the number of tuples in memory. In the SM model, we showed that $C_{IO.spill.inc}(R,S)$ is based on the $S$ tuples in memory. Instead, the last spill for the TS algorithm holds both $S$ and $R$ tuples in memory. For simplicity we make the assumption that memory is used equally by each dataset (in practice the number of tuples from each dataset in memory depends on the relative interval distributions). Hence, the incremental formulas for spilling (whether $R$ or $S$ are spilled respectively) are given by:

$$\begin{aligned} C_{IO.spill.inc}(R,S) &= \frac{\lambda_R}{T_{f.R}} * \frac{\frac{1}{2}T_{M.S}}{\lambda_S} \\ C_{IO.spill.inc}(S,R) &= \frac{\lambda_S}{T_{f.S}} * \frac{\frac{1}{2}T_{M.R}}{\lambda_R} \end{aligned} \tag{4.23}$$

**Number of spill files.** Note that for the SM algorithm, to calculate the number of spills we use the fact that each spill flushes the whole available memory $F_M$. However, since in the TS algorithm, memory is occupied by two datasets, there is an alternating spilling pattern that causes *part of the memory* to be spilled (i.e., $\beta F_M$, where $\beta < 1$). In order to determine $\beta$, consider the sequence of spills. To simplify the process, assume that the arrival rate in both datasets is the same; then at the first spill the memory is equally occupied by the two datasets. This means that the first spill, will free half of the memory ($\frac{1}{2}F_M$, i.e. $\beta_1 = \frac{1}{2}$). If the freed memory is then equally occupied by the two datasets, at the second spill, the largest dataset in memory will have size $\frac{3}{4}F_M$ (i.e., $\frac{1}{2} + \frac{1}{4}$), which is how much memory will be freed (while $\frac{1}{4}F_M$ remains in memory). For the third spill, the empty memory is equally divided among the two relations, adding $\frac{3}{8}F_M$ tuples to the existing $\frac{1}{4}F_M$ tuples; thus the memory to be freed becomes $\frac{5}{8}F_M$. Under these simplifying assumptions, the portion of freed memory from each spill follows the following sequence, starting with the first spill: $\frac{1}{2}$, $\frac{3}{4}$, $\frac{5}{8}$, $\frac{11}{16}$, $\frac{21}{32}$, ... It can be shown that the portion of the freed memory at the $n-th$ split satisfies the following recurrence relation: $\beta_n = 1 - \frac{1}{2}\beta_{n-1}$ for $n = 2, 3, ...$, where $\beta_1 = \frac{1}{2}$. Solving this recurrence, reveals that as $n \to \infty$, $\beta_n \to \beta = \frac{2}{3}$.

The above recurrence formula can easily be generalized to the case where the number of active tuples from each stream is not the same. For example, assume that $L_R > L_S$, then the recurrence becomes: $\beta_n = 1 - \frac{L_S}{L_R+L_S}\beta_{n-1}$, where $\beta_1 = \frac{L_S}{L_R+L_S}$ (since $L_R$ is larger, tuples from relation $R$ will occupy a larger part of the memory; to free this part, we will spill the smaller part currently in memory, which are tuples from $S$). Solving this recurrence, reveals that as $n \to \infty$, $\beta_n \to \beta = \frac{L_S+L_R}{2L_S+L_R}$.

If on average, for each spill we process $\beta F_M$ tuples from a stream, then to process the whole stream of $R$ or $S$, the number of spills required respectively is given by:

$$I_{total}(R) = \left\lfloor \frac{F_R}{\lfloor \beta F_M \rfloor} \right\rfloor$$
$$I_{total}(S) = \left\lfloor \frac{F_S}{\lfloor \beta F_M \rfloor} \right\rfloor \tag{4.24}$$

The increment spill files begin when the end time of $t_{R'}$ (or $t_{S'}$) occurs after the last start time in $S$ (respectively $R$). The cool-down spill files are created from a number of increment spill files up to the size of the steady-state spill file. If the increment spill is continually appended to itself until it reaches the steady state size, the formulas for the number of spills in the cool down phase (whether $R$ or respectively $S$ spills) are given by:

$$I_{coolDown}(R,S) = \left\lfloor \frac{C_{IO.steady.spill}(R,S)}{C_{IO.spill.inc}(R,S)} \right\rfloor$$
$$I_{coolDown}(S,R) = \left\lfloor \frac{C_{IO.steady.spill}(S,R)}{C_{IO.spill.inc}(S,R)} \right\rfloor \tag{4.25}$$

As a result, the number of spills during the steady state phase for relation $R$ (namely, $I_{steadyState}(R,S)$) and for relation $S$ (namely, $I_{steadyState}(S,R)$) are given by:

$$I_{steadyState}(R,S) = I_{total}(R) - I_{coolDown}(R,S)$$
$$I_{steadyState}(S,R) = I_{total}(S) - I_{coolDown}(S,R) \tag{4.26}$$

**Calculating the I/O Cost.** The I/O cost is determined by the number of frames written and read during the time-sweep interval join. Since each spill file is only written once and we spill both relations, the I/O for writes is given by:

$$C_{IO.written}(R,S) = F_R + F_S \tag{4.27}$$

Now consider the read I/O for the steady-state and cool-down spilling phases. The read I/O during the steady-state when $R$ (respectively $S$) is spilled is given by:

$$\begin{aligned} C_{IO.steadyState}(R,S) &= I_{steady}(R,S) * C_{IO.steady.spill}(R,S) \\ C_{IO.steadyState}(S,R) &= I_{steady}(S,R) * C_{IO.steady.spill}(S,R) \end{aligned} \tag{4.28}$$

For our simplified example, the two sides alternate in building the spill files that total the increment spills during the cool-down phase, as shown in Figure 4.5. The cool-down SM model must be split into two summation, one for each dataset. Figure 4.5 shows increment spills alternate between the two datasets, thus cool-down increment count for $R$ and $S$ is split in half and one side will be for the odd number of spills and the other side represents the even number spills. The following formula shows the $C_{IO.coolDown}(R,S)$, if R has been chosen to hold the odd spills.

$$\begin{aligned} C_{IO.coolDown}(R,S) &= C_{IO.spill.inc}(R,S) * \sum_{i=1}^{I_{coolDown}(R,S)} i \\ C_{IO.coolDown}(S,R) &= C_{IO.spill.inc}(S,R) * \sum_{i=1}^{I_{coolDown}(S,R)} i \end{aligned} \tag{4.29}$$

The I/O cost model is the sum of the number of writes and the reads for each spill phase.

$$\begin{aligned} C_{IO} = C_{IO.written}(R,S) & \\ &+ C_{IO.steadyState}(R,S) + C_{IO.steadyState}(S,R) \\ &+ C_{IO.coolDown}(R,S) + C_{IO.coolDown}(S,R) \end{aligned} \tag{4.30}$$

**Calculating the CPU Cost.** The CPU cost can be computed by the size of the result (which we already estimated as $JE(R,S)$ in Section 4.3) plus the estimated number of comparisons that did not produce any result. To estimate the non-result producing comparisons, we note that the TS algorithm only compares active tuples from each relation.

Thus, each time a tuple is added to memory it will match with all tuples in memory. When a tuple becomes inactive, the tuple is removed from memory before it would be used in a unproductive comparison. The CPU cost for TS algorithm's is just the join size estimation, since no unproductive comparisons are made. Thus:

$$C_{CPU} = JE(R, S) \tag{4.31}$$

During the spilling phase, the algorithm uses a spilling method similar to SM and must add the unproductive comparisons during the spilling phase. During spilling, an unproductive comparison is made for each tuple in memory as it scans through the spill file to find all matches.

$$C_{CPU.unproductive}(R, S) = \beta * T_M * I_{steady}(R, S) * \lambda_R + \beta * T_M * I_{steady}(S, R) * \lambda_S \tag{4.32}$$

Resulting formula for the CPU for spilling cases:

$$C_{CPU.spill} = JE(R, S) + C_{CPU.unproductive}(R, S) \tag{4.33}$$

### 4.4.3 Forward-Scan Interval Join Model

The Forward-Scan Interval Join (FS) is a cross between SM and TS. The algorithm splits memory between the two relations similar to TS but picks the next tuple to process based on time order like SM. FS starts by picking the first time ordered tuple from either relation and loads all matching tuples from the other relation into memory. Assume that the first tuple is from $R$ ($t_R$), then all overlapping tuples from $S$ are loaded into memory. These are all tuples from $S$ that start during the time interval of $t_R$, i.e., $\lambda_S d_R$ tuples. The process continues by selecting the next time ordered tuple to process. If this next tuple is from $S$ ($t_S$), then all overlapping tuples from $R$ are loaded into memory; these are $\lambda_R d_S$ tuples. That is, memory would then hold all previous tuples from $S$ ($\lambda_S d_R$) and the new $R$ tuples ($\lambda_R d_S$). If during this process memory becomes full, the algorithm must spill. To perform the join, the FS algorithm reserves the following five frames: a frame for each of the two input streams $R$ and $S$, one frame for the output result, and one frame for each of the spilled datasets for $R$ and $S$. In this case, the FS algorithm will spill if the following formula is true:

$$\frac{\lambda_R d_S}{T_{f.R}} + \frac{\lambda_S d_R}{T_{f.S}} > F_M - 5 \tag{4.34}$$

Note, that spilling can also happen using one relation in memory, for example, if all overlapping $S$ tuples for $t_R$ cannot fit in memory (i.e., $\frac{\lambda_S d_R}{T_{f.S}} > F_M - 5$). Similarly, in the case that the first tuple comes from $S$ and the overlapping $R$ tuples cannot fit in memory (i.e., $\frac{\lambda_R d_S}{T_{f.R}} > F_M - 5$).

More interesting is the case where spilling happens while having loaded tuples from both relations in memory. Following the above example assume that when the memory gets full the relative portions from the two relation tuples currently in memory satisfy: $\frac{\lambda_S d_R}{T_{f.S}} > \frac{\lambda_R d_S}{T_{f.R}}$. That is, during the first spill, $S$ happens to occupy more memory (in practice this would be based on the data properties and arrival rates). The FS algorithm will pause and try to free the larger relation currently in memory ($S$). To do this, the algorithm needs to find all matches with $R$ tuples (whether in memory or from the $R$ stream) for the memory resident $S$ tuples. Considering Figure 4.6, assume that the current $S$ tuples in memory start from tuple $t_S$ until tuple $c$. Any tuples accessed from the $R$ stream starting with tuple $t_{R'}$ need to spill since they may match with future $S$ tuples. After all such matchings are found, memory flushes the $S$ tuples and the FS algorithm restarts with the next tuple in time order. Note that since the $S$ tuples in memory have been processed, time has advanced for the $S$ relation and the next $S$ time ordered tuple comes from the $S$ stream starting at $t_{S'}$ while the next time ordered tuple from $R$ remains tuple $t_R$, which is still in memory and may have more $S$ matches. For this tuple $t_R$, we need to load new $S$ tuples that may match with it, starting with $t_{S'}$. Since time has advanced in $S$ the next time ordered tuple will be from $R$ until time advances in relation $R$ and catches up with $S$. This means that more $S$ tuples will be loaded to memory from $S$. Under the simplifying assumptions of similar arrival rates and equal interval sizes, this will lead to the previous spilling scenario, i.e. the same relation (in our example $R$) will be spilled. In practice the spilling order will change if there is a long temporal distance between two consecutive tuples in $S$ which allows tuples in $R$ to be processed without loading new matching tuples from $S$, thus effectively, 'restarting' the spilling pattern.

Similarly with the SM algorithm, the FS algorithm includes three phases during a spill: warm-up, steady-state, and cool-down. The I/O cost model is calculated by determining the size of each spill, counting the number of spills, and then building a full I/O cost formula. Below we make the simplifying assumption that the same relation will always spill.

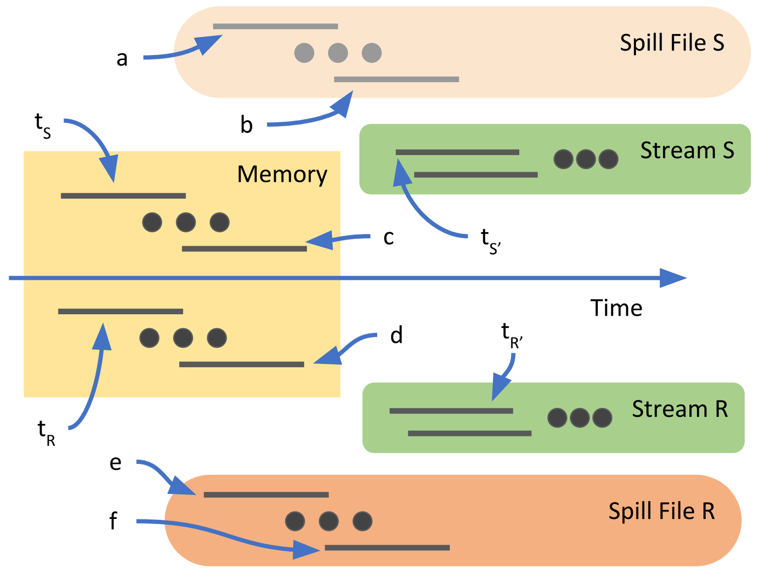


Figure 4.6: Important tuples during the spilling process for the forward-scan interval join. The colored blocks represent the intervals in memory, the input streams for $R$ and $S$ and the spill files for $R$ and $S$. Tuple $t_S$ corresponds to the first $S$ tuple currently in memory, tuple $c$ is the last $S$ tuple in memory, tuple $a$ is the first tuple in the spill file of $S$, tuple $b$ corresponds to the last tuple in that spill file, and tuple $t_{S'}$ is the next tuple from the $S$ stream. Similarly, tuple $t_R$ is the first $R$ tuple currently in memory, tuple $d$ corresponds to the last $R$ tuple in memory, tuple $e$ is the first tuple in the spill file of $R$, tuple $f$ corresponds the last tuple in that spill file, and tuple $t_{R'}$ is the next tuple from the $R$ stream. In our example, spill file $S$ is not used.

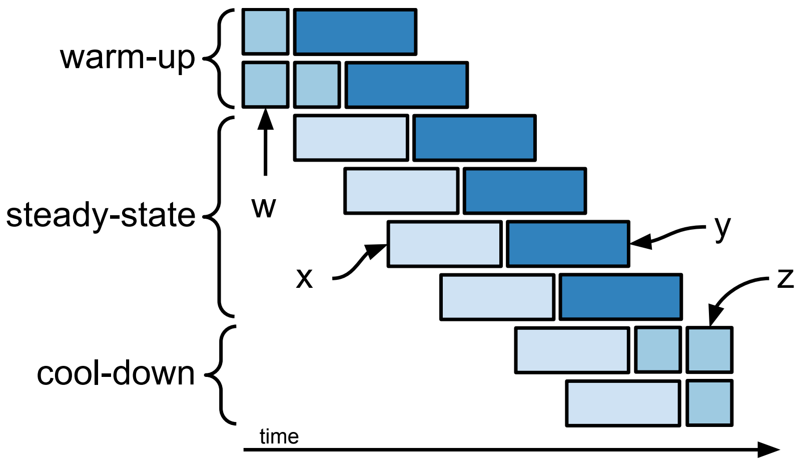


Figure 4.7: The $R$ spilling phase for the Forward Scan interval join. The FS spill files have been split into two parts: the part from the start of $t_R$ until the start of tuple $c$ (before $c$), and the interval of tuple $c$ (during $c$). The steady-state spill file depicts these two parts as $x$ for before $c$ and $y$ for during $c$. The warm-up and cool-down spill phases also utilize an increment spill file: depicted as $w$ (in the warm-up) and $z$ (in the cool-down).

**Size of the spill files.** The spill size is determined by the properties of the tuples being freed from memory. Recall that in our example, the earliest time currently in memory is the start of tuple $t_R$. Let $c$ be the tuple from $S$ with the latest end point (see Figure 4.6). When the FS algorithm pauses due to memory becoming full, the $S$ tuples in memory must be matched with all $R$ tuples after $t_R$ (from the memory or form the stream). Note that all $R$ tuples before $t_R$ have already been processed. The join process will continue to match $R$ tuples starting from the start of $t_R$ until the end of the $c$ tuple. This interval can be divided into two parts: (i) from the start of $t_R$ until the start of tuple $c$, and (ii) the interval of tuple $c$. To calculate the size of the spill we need to find how many $R$ tuples are in these two parts.

Consider the steady-state spill phase for the first of the above intervals, shown in Figure 4.7 as $x$. There are two cases depending on whether the memory is filled only by $S$

tuples or if the memory is shared between $R$ and $S$ tuples. In the case where memory is full with $S$ tuples, these tuples were loaded because of overlapping with $t_R$ (which has size $d_R$).

Thus the spill file, contains those $R$ tuples that arrive during $d_R$; the number of frames to hold these $R$ tuples is thus: $(\frac{d_R * \lambda_R}{T_{f.R}})$. The other steady-state scenario considers the case when memory is occupied by tuples from both relations. In this scenario, the spill size should be reduced by the number of $R$ tuples that are already in memory; such tuples can stay in memory and will be matched with future $S$ tuples, i.e., this part does not need to be spilled. The number of $R$ tuples that are in memory has the following size in frames: $F_M - \frac{d_S * \lambda_S}{T_{f.S}}$. The resulting steady-state formula for these two scenarios is:

$$C_{IO.beforeC}(R,S) = \begin{cases} \dfrac{d_R * \lambda_R}{T_{f.R}}, & \text{if } \dfrac{d_R * \lambda_S}{T_{f.S}} > F_M \\ \dfrac{d_R * \lambda_R}{T_{f.R}} - (F_M - \dfrac{d_S * \lambda_S}{T_{f.S}}), & \text{otherwise} \end{cases} \tag{4.35}$$

Consider now the steady-state spill phase for the second part of the spill, the time interval during tuple $c$, depicted as the $y$ spill file in Figure 4.7. In this case, the spill file is the number of frames that hold the $R$ tuples which start during the interval of tuple $c$ (which has size $d_S$):

$$C_{IO.duringC}(R,S) = \frac{d_S * \lambda_R}{T_{f.R}} \tag{4.36}$$

Next consider the warm-up spill phase shown in Figure 4.7. Note the warm-up spill phase is only needed when the spill part coming from $R$ tuples that arrive before the start of the $c$ tuple, is smaller than the steady-state spill file. This occurs only when memory is filled by a single relation, in our case $S$. The spill increment file (depicted as $w$ in Figure 4.7) is used to calculate the amount of $S$ tuples that can be processed in one batch of memory for the "before tuple $c$" spill part. As the figure suggests, each new spill file increases in size by one spill increment file during the warm-up spill phase until it reaches the steady-state spill file size. Consider the first spill that happens in the warm-up phase. The $R$ tuples that arrive before tuple $c$ will be matched against the tuples of $S$ in memory. Currently, there are $T_{M.S}$ tuples from $S$. These tuples arrived during an interval of length $\frac{T_{M.S}}{\lambda_S}$. During this interval, the number of $R$ tuples that arrived is: $\frac{T_{M.S}}{\lambda_S} * \lambda_R$. The size of the spill $C_{IO.spill.inc}(R, S)$, in number of frames, is thus:

$$C_{IO.spill.inc}(R, S) = \frac{T_{M.S}}{\lambda_S} * \frac{\lambda_R}{T_{f.R}} \tag{4.37}$$

Note that the resulting $C_{IO.spill.inc}(R, S)$ formula is the same as the SM spill increment shown in Formula 4.9. Similarly to the warm-up spill phase, the cool-down spill phase uses the increment spill file (depicted as $z$ in Figure 4.7) to decrease the "during tuple $c$" spill part (shown as $y$ in Figure 4.7).

**Number of spill files.** The next step is to determine the number of spills for each of the three phases: warm-up, steady state, and cool-down. First consider the total number of spills that occur during the join. To enter a spill process, memory will be full; assume that $S$ has more tuples in memory than $R$. There are two scenarios: either $S$ fills the full memory ($F_{M.S}$) or both relations fill the memory and only the $S$ part of memory ($\frac{d_R * \lambda_S}{T_{f.S}}$) is processed. The total number of $S$ frames ($F_S$) is divided by these batches of $S$

tuples to determine the total number of spills. Since the last pass could be done in memory the total number is reduced by one. Thus:

$$I_{total}(R,S) = \frac{F_S}{min(F_{M.S}, \frac{d_R*\lambda_S}{T_{f.S}})} - 1 \tag{4.38}$$

The number of warm-up spills is determined by how many times it would take to build the steady-state spill file $C_{IO.beforeC}(R,S)$ using the increment spill file, i.e.:

$$I_{warmUp}(R,S) = \left\lfloor \frac{C_{IO.beforeC}(R,S)}{C_{IO.spill.inc}(R,S)} \right\rfloor \tag{4.39}$$

Similarly the number of spills in the cool-down phase is given by:

$$I_{coolDown}(R,S) = \left\lfloor \frac{C_{IO.duringC}(R,S)}{C_{IO.spill.inc}(R,S)} \right\rfloor \tag{4.40}$$

The number of spills in the steady-state is then given by:

$$I_{steady}(R,S) = I_{total}(R,S) - I_{warmUp}(R,S) - I_{coolDown}(R,S) \tag{4.41}$$

**Calculating the I/O Cost.** The I/O cost is determined by the number of frames written and read during the FS algorithm. Since the FS algorithm only spills one relation ($R$) and is only written once, the I/O for writes is given by:

$$C_{IO.write}(R,S) = F_R \tag{4.42}$$

Now consider the read I/O for the steady-state, warm-up and cool-down spill phases. The steady-state spilling phase I/O is calculated by adding the spilled frames from each steady-state spill make up of both the “before tuple $c$” and “during tuple $c$” spill parts. Given by:

$$C_{IO.steady}(R,S) = I_{steady}(R,S) * (C_{IO.beforeC}(R) + C_{IO.durringC}(R)) \tag{4.43}$$

Figure 4.7 depicts the parts of each spill file. The warm-up spilling phase I/O includes the steady-state spill part for “during tuple $c$” (depicted as $y$) and the increasing number of increment spills (depicted as $w$). Thus:

$$\begin{aligned} C_{IO.warmUp}(R,S) = I_{warmUp}(R,S) * C_{IO.duringC}(R,S) \\ + C_{IO.spill.inc}(R,S) * \sum_{i=1}^{I_{warmUp}(R,S)} i \end{aligned} \tag{4.44}$$

In this summation, the case i=1 corresponds to reading the first spill file which has the size $C_{IO.spill.inc}(R,S)$. Then this spill file is incremented by $C_{IO.spill.inc}(R,S)$ to create the second spill file, etc. until $i = I_{warmUp}(R,S)$.

Figure 4.7 the cool-down phase and show the two parts: the increment spill part (depicted as $z$) and the steady-state spill for the “before tuple $c$” part (depicted as $x$). In the cool-down spilling phase we start from the last spill which has size $C_{IO.spill.inc}(R,S)$. Then continue to the second to last spill, incrementing the spill size similar to the warm-up spill phase. Thus:

$$\begin{aligned} C_{IO.coolDown}(R,S) = I_{coolDown}(R,S) * C_{IO.beforeC}(R,S) \\ + C_{IO.spill.inc}(R,S) * \sum_{i=1}^{I_{coolDown}(R,S)} i \end{aligned} \tag{4.45}$$

The I/O cost model is the sum of the frames written and read during the three spilling phases. Thus:

$$C_{IO}(R,S) = C_{IO.write}(R,S) + C_{IO.warmUp}(R,S) + C_{IO.steady}(R,S) + C_{IO.coolDown}(R,S) \tag{4.46}$$

**Calculating the CPU Cost.** The CPU cost can be computed by the size of the result (which we already estimated as $JE(R,S)$ in Section 4.3) plus the estimated number of comparisons that did not produce any result. To estimate the non-result producing comparisons, we note that the FS algorithm will stop comparing $t_R$ with $S$ on the first $S$ tuple that starts after $t_R$'s end time (i.e., one non-result comparison per $R$ tuple). When $t_R$ reaches the end of the relation's time interval, no non-producing comparisons will be made and can be removed $(\lambda_R * (d_R - 1))$. The same is true for when the selected tuple comes from the $S$ relation. The total CPU cost is thus:

$$C_{CPU}(R,S) = JE(R,S) + T_R - \lambda_R * (d_R - 1) + T_S - \lambda_S * (d_S - 1) \tag{4.47}$$

### 4.4.4 Overlap Interval Partition Join Model

The Overlap Interval Partition Join (OIP) is the first of our two partitioning interval joins. These joins utilize the partitions defined by their algorithm as the spill files. Thus the model presented here considers how many partitions are created and their sizes to determine the I/O cost.

To determine when the OIP algorithm spills data, consider the two activities used to complete the join process: *partitioning* and *joining*. The partitioning activity evenly splits the temporal range (covered by the intervals of a relation) into slots. The OIP uses these slots to create an overlapping temporal partitioning scheme, where each partition is defined by its start and end slot. An example scenario with three slots was discussed in Figure 3.13; here each temporal partition has been assigned an identifier, composed by a start slot and followed by an end slot. A partition will store all relation tuples whose intervals start and end times fall within the respective slots. The aim of the partitioning activity is to assign tuples of each relation to their respective partitions.

As with the other algorithms, prior to joining, the data is assumed sorted, although in OIP's case, the sorting keys are different. Instead of sorting tuples by their interval start and end times, the sorting process in the OIP algorithm uses each tuple's starting and the ending slots. The tuples are sorted first in increasing order of their ending slot, followed by the decreasing order of their starting slot; these slots are easily computed as a tuple is read using the tuple's start and end times. Effectively, this slot-based sort order creates the overlapping temporal partitions. That is, the partitions can be created by reading the sorted stream of each relation. Looking at Figure 3.13 the ordered stream of a relation will have first the tuples from partition (1,1) followed by the tuples from partition (2,2), (1,2), (3,3), (2,3) and (1,3). Note that within each partition the tuples are not ordered. Further, there may be partitions without tuples; the algorithm below simply reads the next tuple from the relation stream. Below we assume for simplicity that both relations use the same slots, resulting to the same respective partitions.

OIP is a specialized block nested-loop algorithm therefore we will use one frame in memory to read tuples from the outer relation (say $R$), one frame for storing the join results and the rest of the memory frames will have tuples from the other relation ($S$). The joining activity begins by filling up the available memory with tuples from the $S$ stream (that is, partitions $S_{(1,1)}, S_{(2,2)}, S_{(1,2)}, ...$). We then read the first frame from relation $R$ (starting from $R_{(1,1)}$). For each tuple from $R$ the algorithm will calculate which $S$ partitions are overlapping with it and will proceed to join that tuple with tuples from the overlapping $S$ partitions.

In general, since the OIP algorithm makes a single pass over $S$ by loading it in blocks from its stream, there may be other partitions from $S$ also in memory interleaved with the needed partitions. For example, if we are processing a tuple from $R_{(2,2)}$ the OIP will need in memory all partitions from $S_{(2,2)}$ until $S_{(1,3)}$. This includes partition $S_{(3,3)}$ which is loaded from the stream but is not needed for this join. Such interleaved $S$ partitions occupy memory space, thus it may happen than not all needed partitions for the specific $R$ tuple are in memory. If an overlapping partition from $S$ could not be fully loaded into memory, the algorithm will pause and process all $S$ tuples currently in memory, by reading tuples from the $R$ stream. Since these $R$ tuples may match with future tuples from $S$, they are saved to a spill file for $R$.

The worst case occurs when processing tuples from partition $R_{(1,1)}$: using the above example, the OIP algorithm will require in memory all the related overlapping $S$ partitions, namely, $S_{(1,1)}$, $S_{(1,2)}$, and $S_{(1,3)}$. Given the ordering of the $S$ stream, the OIP join algorithm will need to load in memory the whole $S$ relation (since partition $S_{(1,3)}$ is at the end of the stream based on the above ordering). In this worst-case scenario, spilling occurs if the stream $S$ is not able to fit into memory, that is:

$$F_S > F_M + 2 \tag{4.48}$$

To simplify calculating the I/O and CPU cost for the OIP algorithm, the temporal partitions are grouped by slot length. An example for the partition slot groups of $S$ is shown in Figure 4.8. Here slot groups are identified by the relation and partition's slot length. For example, the $S$ partitions that have length one slot are in the group $sl_1^S$, which in our example includes partitions $S(1,1)$, $S(2,2)$, and $S(3,3)$. Similar groups will be created for $R$. We assume that each partition within a slot group has a similar number of tuples.

The I/O and CPU cost models consider the join of all slot group pairs from $R$ and $S$. The models use the number of overlapping partitions to determine the number of times these partitions in a slot group will be joined. We start with a method to determine the number of joined partitions for a pair of slot groups. Then we discuss how to count the number of tuples in a partition and how to determine the number of frames in a partition. Finally formulas for the overall I/O and CPU costs are presented.

**Number of partition joins.** The next formula considers the number of joins between individual $R$ partitions in one slot group and individual $S$ partitions in another slot group. Figure 4.9 depicts an example with $k = 6$ slots and considers the joining of slot groups $sl_3^R$ and $sl_2^S$. In general the number of partitions in a slot group $sl_i$, denoted by $|sl_i|$ is given by $|sl_i| = k - i + 1$. Thus the $sl_3^R$ slot group has 4 partitions. Consider the number of $sl_2^S$ partitions that overlap a given $sl_3^R$ partition, say $R(2,4)$ take from the middle of the

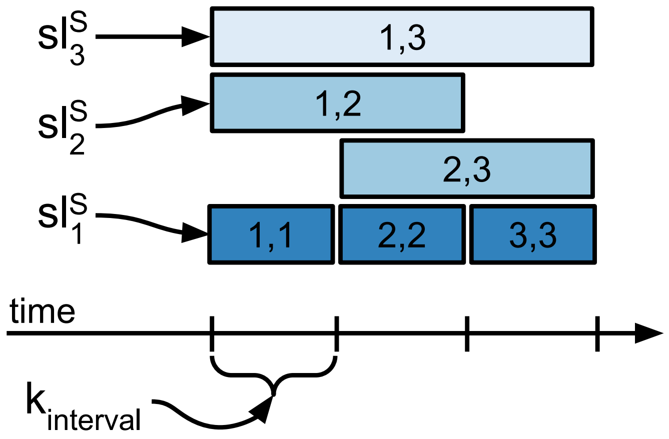


Figure 4.8: The slot partition groups for $S$.

partition group, the total number of overlapping partitions given by: $sl_3^R + sl_2^S - 1 = 3+2-1$; this is computed similarly with how the join size estimation computes overlap between two tuples. Recall from Section 4.3 that the overlap was calculated by counting: (i) the number of tuples that end during a tuple $t_R$ and, (ii) all the tuples that start during tuple $t_R$. For the partitioning example, partition $R(2,4)$ overlaps four $sl_2^S$ partitions, namely: $S(1,2)$ which ends in the partition, and $S(2,3)$, $S(3,4)$, and $S(4,5)$ partitions which start during the $R$ partition. The total number of joined partitions between slot groups $sl_i^R, sl_j^S$ is given by the number of partitions in the $sl_i^R$ group multiplied by the number of overlapping $S$ partitions in the $sl_j^S$ group, namely:

$$I_{total}(sl_i^R, sl_j^S) = |sl_i^R| * (sl_i^R + sl_j^S - 1) \tag{4.49}$$

However, similar to the join size overestimation, this approach over counts the joined partitions at the beginning and end of a relation's time range. Consider partition

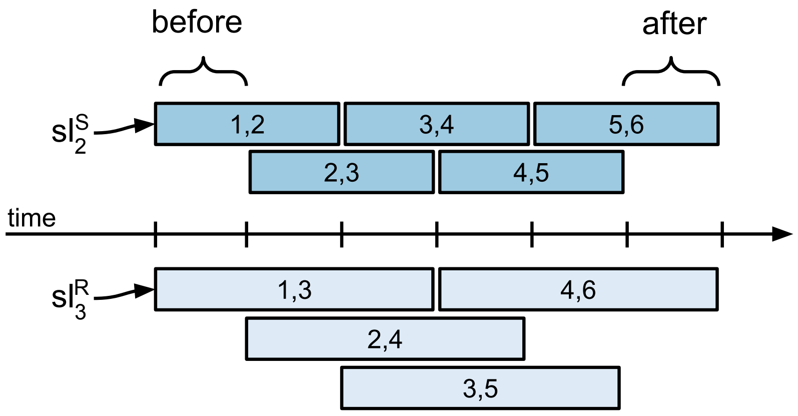


Figure 4.9: An example join where $k$ has been set to 6 and a pair of slot groups are being joined: $sl_3^R$ and $sl_2^S$. The before and after phases where the number of overlapping partitions changes due to partitions starting before the first slot or ending after the last slot.

$R(1,3)$; the previous discussion suggests that it should overlap four partitions, but it only overlaps three, namely: $S(1,2)$, $S(2,3)$, and $S(3,4)$. That is, partition $S(0,1)$ should not be counted as it starts before the first $R$ slot. The number of such over-counted partitions can be estimated using an approach similar to the "before" formula in the join size estimation (Section 4.3). The formula has been updated to count partitions instead of individual tuples. In particular, updating the $JE_{before}(R,S)$ formula to count the partitions involves: (i) setting $\lambda_R = 1$, since only one partition starts at each slot (similarly for $\lambda_S$) and (ii) replace $d_R$ by the slot length. The resulting overestimations are:

$$I_{before}(sl_i^R, sl_j^S) = I_{after}(sl_i^R, sl_j^S) = \sum_{u=1}^{sl_j^S - 1} u \tag{4.50}$$

Hence, the number of joined partitions after removing the 'before' and 'after' overestimation from the total join estimation is:

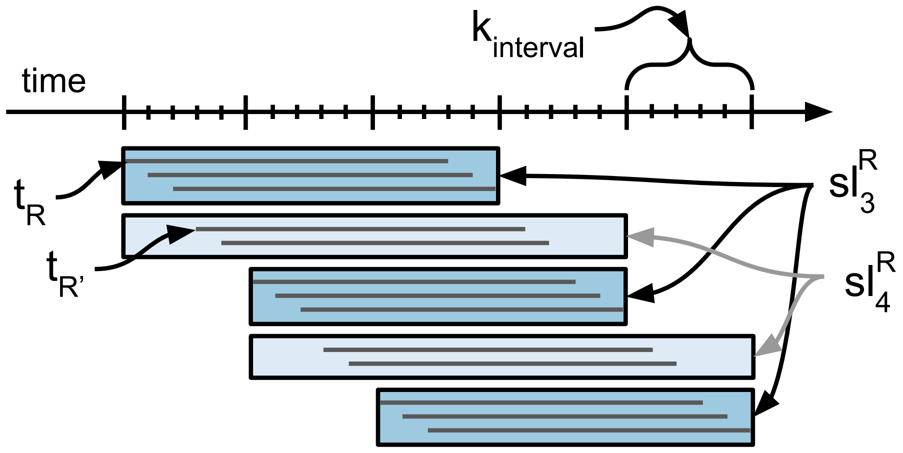


Figure 4.10: The resulting partitioning over four slots on $R$ relation with these properties: $k_{interval} = 5$, $d_R = 13$, and $\lambda_R = 1$.

$$I_{join}(sl_i^R, sl_j^S) = I_{total}(sl_i^R, sl_j^S) - I_{before}(sl_i^R, sl_j^S) - I_{after}(sl_i^R, sl_j^S) \tag{4.51}$$

**Number of tuples in a partition.** Figure 4.10 depicts a time range with 25 time instances, divided among an $k = 5$ slots. The example relation $R$ has the following properties: each time instance there is one tuple arriving ($\lambda_R = 1$) while the interval length of the each tuple is $d_R = 13$. The figure highlights two cases for how tuples will be partitioned among these slots, using tuples $t_R$ and $t_{R'}$ as examples. Depending on when a tuple starts, it can in this case be assigned to a partition that covers three (as for $t_R$) or four ($t_{R'}$) slots. Since each slot has the same size, the time interval covered by a slot in $R$ is:

$$K_{interval}(R) = \left\lceil \frac{r_{R.end} - r_{R.start}}{k} \right\rceil \tag{4.52}$$

A tuple with length of $d_R$ will be assigned to a partition that covers $\left\lceil \frac{d_R}{K_{interval}(R)} \right\rceil$ or $(\left\lceil \frac{d_R}{K_{interval}(R)} \right\rceil + 1)$ slots (depicted as $sl_3^R$ and $sl_4^R$ respectively). In the first case, the partition contains $\left\lceil \frac{d_R}{K_{interval}(R)} \right\rceil * K_{interval}(R)$ time instants (and there are $\lambda_R$ tuple arrivals per time instant). However, the partition only holds tuples that start in the first slot and end in the third slot. The tuple length ($d_R$) is subtracted from the partition's total time interval to find the number of time instances when a tuple will start during the first partition, $Q(R) = ((\left\lceil \frac{d_R}{K_{interval}(R)} \right\rceil * K_{interval}(R)) - d_R - 1)$. The tuples in the partition with length $(\left\lceil \frac{d_R}{K_{interval}(R)} \right\rceil + 1)$ slots, start in the remaining time instances of the first slot, that is: $(K_{interval}(R) - Q(R))$. To find out the number of tuples in each case, we multiply by the arrival rate of $R$, $\lambda_R$:

$$T_{partition}(sl_i^R) = \begin{cases} Q(R) * \lambda_R, & \text{if } sl_i^R = \left\lceil \dfrac{d_R}{K_{interval}(R)} \right\rceil \\ (K_{interval}(R) - Q(R)) * \lambda_R, & \text{if } sl_i^R = \left\lceil \dfrac{d_R}{K_{interval}(R)} \right\rceil + 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.53}$$

The size of an individual slot partition in frames is then:

$$C_{IO.partition}(sl_i^R) = \left\lceil \frac{T_{partition}(sl_i^R)}{T_{f.R}} \right\rceil \tag{4.54}$$

**Calculating the I/O Cost.** The I/O cost is determined by the number of partitions written and read to process the join. OIP writes the $R$ partitions only once and processes the join with a single pass over $S$ during which, $R$ may be read multiple times. Thus the number of writes is:

$$C_{IO.write}(R,S) = F_R \tag{4.55}$$

Recall that all $S$ partitions are loaded into memory. If an individual $S$ partition cannot fit into memory multiple passes over $R$ (denoted as $I_{passes}$) are needed to find all matches in $S$. $I_{passes}$ is determined by how many blocks of size $F_M$ we can have in this $S$ partition:

$$I_{passes}(sl_j^S) = \left\lceil \frac{C_{IO.partition}(sl_j^S)}{F_M} \right\rceil \tag{4.56}$$

Note that $I_{passes}(sl_j^S)$ is looking at an individual $S$ partition. A slot group would need to make $I_{passes}(sl_j^S)$ over each slot partition in the group.

Join partition count is made up of joining all slot groups from each relation. The set of $R$ slot groups is expressed as $\langle sl^R \rangle$ which represents $sl_1^R$, $sl_2^R$, and $sl_3^R$ from Figure 4.8. Note that slot groups may not have tuples located in their individual partitions, in this case those slot groups would be left out of the set. The I/O cost includes the partitions written and the summation of joining all pairs of partition groups. For each pair the number of partition joins ($I_{join}(sl_i^R, sl_j^S)$) is multiplied by the number of passes over $S$ needed to process the individual partition $R$ and the size of the $R$ partition.

$$\begin{aligned} C_{IO}(\langle sl^R \rangle, \langle sl^S \rangle) &= C_{IO.write}(R,S) \\ &+ \sum_{i=1}^{n} \sum_{j=1}^{m} I_{join}(sl_i^R, sl_j^S) * I_{passes}(sl_j^S) * C_{IO.partition}(sl_i^R) \end{aligned} \tag{4.57}$$

where $n$ is the size of $\langle sl^R \rangle$ and $m$ is the size of $\langle sl^S \rangle$

**Calculating the CPU Cost.** The CPU cost model follows the I/O cost model but changes the number of frames read for the number of comparisons. OIP uses a nested loop to join individual partitions so the number of comparisons is simply the number of tuples in $R$ partitions times the number of tuples in the $S$ partition. Thus:

$$C_{CPU}(\langle sl^R \rangle, \langle sl^S \rangle) = \sum_{i=1}^{n} \sum_{j=1}^{m} I_{join}(sl_i^R, sl_j^S) * T_{partition}(sl_i^R) * T_{partition}(sl_j^S) \quad \text{where } n \text{ is the size of } \langle sl^R \rangle \text{ and } m \text{ is the size of } \langle sl^S \rangle \tag{4.58}$$

## 4.4.5 Disjoint Interval Partitioning Model

The Disjoint Interval Partitioning Join (DIP) is the second partitioning algorithm. The DIP algorithm assumes the input data is time ordered similarly to the SM, TS and FS algorithms. The algorithm is split into two activities: *partitioning* and *joining*. The partitioning activity partitions each relation separately. Assume that the DIP algorithm first partitions relation $R$. It starts by selecting the first $R$ tuple and assigning to a partition. A heap maintains the end time of the last tuple added in each partition. For each next $R$ tuple selected there are two options: assign it to an existing partition or create a new partition. If the start time of the new $R$ tuple is greater or equal to the smallest end time in the heap, it is added to an existing partition. Otherwise (i.e. it is less), the tuple overlaps all existing partitions and must be added to a new partition. The partitioning process repeats until all $R$ tuples have been assigned a partition. While partitioning the relation, the DIP algorithm will spill if an $R$ tuple cannot be added in memory. The partitioning activity uses two frames that reduce the total memory available: one frame for the input stream and one frame for the output stream. Thus, the DIP algorithm will spill while partitioning $R$ if:

$$F_R > F_M + 2 \tag{4.59}$$

If relation $R$ cannot fit in memory, the partitioning activity switches to utilize one memory frame for each partition. If there are enough frames for the partitions, the partitioning will finish in one pass over $R$. If however, there are more partitions than available frames in memory, the output frame is used to create a spill that will contain all tuples that have not been partitioned (i.e. cannot be assigned to existing partitions). Once the stream of $R$ has been fully processed, the partitioning activity will pick up the spill file with tuples that do not have a partition and repeat the partitioning process again with these tuples as input (effectively making multiple passes over $R$). Following the completion of partitioning relation $R$, all partitions are written to disk; partitioning proceeds with relation $S$. The end of partitioning results in (disjoint) interval partitions for relations $R$ and $S$ (their partitions are depicted respectively as $z$ and $y$ in Figure 4.11).

When partitioning $S$, there are two cases: either the $S$ partitions fit in memory or the $S$ partitions spill to disk. In the first case, the join can be completed with one pass over $R$ (one partition at a time).

In the second case, the join activity will continue by reading relation $S$ from disk using one memory frame per $S$ partition. The join activity picks an $R$ partition and loads the first frame from that partition; it then does a merge-join with all $S$ partitions (one frame per partition at a time). If the number of $S$ partitions are larger than the number of available memory frames, then multiple passes over $S$ will be made when joining with each $R$ partition. The join activity does not backtrack during the merge-join process (since

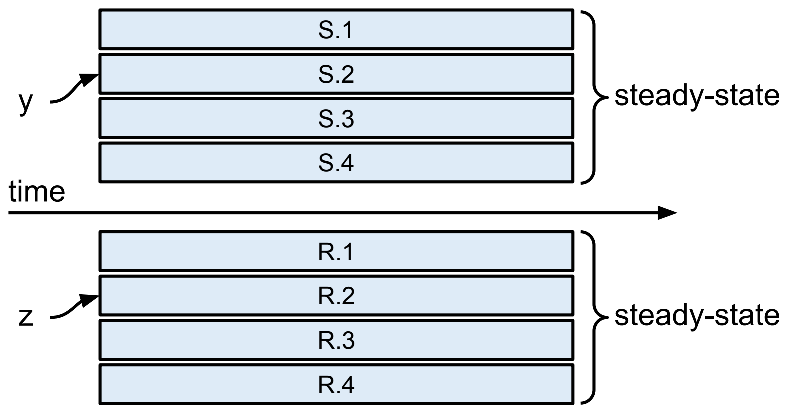


Figure 4.11: The $R$ and $S$ spilling phase for disjoint interval partition join. The spill file denoted as $y$ represents the disjoint partitions in $S$ and $z$ represents the disjoint partitions in $S$.

partitions within a relation have disjoint intervals), but may reread partitions to complete the join with other partitions.

The DIP algorithm does not have warm-up or cool-down spill phases, only the steady-state phase. Similar with the OIP algorithm, to calculate the I/O cost we need to determine the number of partitions and the size of each partition. The two algorithms defer on how the partitions are created and used by the join algorithm.

**Number of partitions.** Recall that each disjoint interval partition has no overlapping intervals. Thus, $I_{total}(R)$ the number of $R$ partitions, is equal to the largest number of active $R$ tuples at a given instance; assuming the simplifying scenario with evenly distributed tuples over the relation $R$, this is given by $L_R$, i.e.:

$$I_{total}(R) = L_R \tag{4.60}$$

**Size of partitions.** Continuing with our example, the (average) size of each partition is simply the size of the relation divided by the number of partitions.

$$C_{IO.partition}(R) = \left\lceil \frac{F_R}{I_{total}(R)} \right\rceil \tag{4.61}$$

**Calculating the I/O Cost.** The I/O cost is determined by the number of frames written and read during the DIP algorithm. Since each partition is only written once and we spill all partitions from both relations, the I/O for writes is given by:

$$C_{IO.write}(R,S) = F_R + F_S \tag{4.62}$$

Consider the scenario where memory does not have enough available frames to hold all $S$ partitions. The algorithm must process batches of $S$ to complete the join with partitions from $R$. The number of batch is based on the number of times portions of $S$ will be loaded into memory. If one frame is designated for $R$ partitions, then available memory is $F_M - 1$. The total number of partitions for $S$ is found using the number of partitions ($I_{total}(S)$) times the size of each partition ($C_{IO.partition}(S)$). The batches to process relation $S$ is the number of frames used to store the $S$ partitions divided by the available memory, given by:

$$I_{batch}(S) = \left\lceil \frac{I_{total}(S) * C_{IO.partition}(S)}{F_M - 1} \right\rceil \tag{4.63}$$

An $R$ partition is read through one frame in memory and must be merged with each batch of $S$ partitions in memory. When it is finished, the next $R$ partition is streamed through this frame. For each $R$ partition, the join will make one pass over all $S$ partitions loaded into memory. The I/O cost for this case includes the frames written for both relations, the frames read for each $R$ partition once for each batch of $S$ partitions, and the frames read for each $S$ partition. Thus:

$$\begin{aligned} C_{IO}(R,S) = C_{IO.write}&(R,S) \\ &+ I_{batch}(S) * I_{total}(R) * C_{IO.partition}(R) \\ &+ I_{total}(S) * C_{IO.partition}(S) \end{aligned} \tag{4.64}$$

**Calculating the CPU Cost.** The CPU cost can be computed by the size of the result (which we already estimated as $JE(R,S)$ in Section 4.3) plus an estimate for the number of comparisons that did not produce any results. Consider when the merge join will make an non-result comparison for each tuple in relation $R$, $T_R$. The DIP merge join selects a tuple from $R$, say $t_R$, and matches it with the selected tuple from each $S$ partition. The comparison will determine if there is a result. The next $R$ or $S$ tuple to be advanced is picked based on which tuple has the smaller end time instance. If the $t_R$ tuple has the smaller time instance, then $t_R$ tuple advances and this $t_R$ tuple will not make any more comparison for with this $S$ partition. If the $S$ tuple has the smaller time instance, then $S$ tuple advances to the next tuple in the partitions and the check is made again. This can lead to a non-result comparison. Consider our simplified relation with a steady arrival rate and consistent time interval for each tuple, the $S$ tuple will only be advanced to a non-result comparison when the tuples share the same end time. The number of non-result comparisons for each $R$ tuple is how many tuples share the $R$ tuples end time, $\lambda_S$. Thus, the CPU cost is the join size estimation and the non-result comparisons:

$$C_{CPU}(R,S) = JE(R,S) + T_R * \lambda_S \tag{4.65}$$

### 4.4.6 Expanding the Models for Complex Datasets

The models described previously have focused on the simpler case where the data within a relation follows the same arrival rate and has the same interval length. This simplistic assumption allowed for the creation of the basic models for spilling, I/O and CPU costs. However, more likely, a real dataset may have various types of interval data. For example, log data tracking jobs from a distributed computing system, may include short jobs that occur frequently and longer jobs that have a lower frequency. Our models can be expanded to support these more complex datasets, by considering a dataset as the union of datasets, each with its own tuple arrival rate and tuple interval length. For example, the short and longer jobs could be separated into two individual datasets: one capturing the behavior of the short jobs and another for the long jobs. In general, we assume that a relation $R$, can be represented as the *union* of $n$ individual subrelations $R_1$, $R_2$,...,$R_n$, where each subrelation $R_i$ is a dataset that has its own fixed interval duration $d_i$ and fixed arrival rate $\lambda_i$ (we assume here that the arrival rate $\lambda_i$ remains the same for the tuples of subrelation $R_i$ over the whole time range of $R$). That is: $R = R_1 \cup R_2 \cup ... \cup R_n$.

Using the property that the Cartesian product is *distributive over union* (that is, $A \times (B_1 \cup B_2) = (A \times B_1) \cup (A \times B_2)$), it follows that the join of two relations $R, S$ each consisting of unions of subrelations, is the union of the joins of the individual subrelations. Each subrelation join (i.e., $R_i \bowtie S_j$) follows the previous models since it uses specific interval lengths (respectively $d_i, d_j$) and arrival rates ($\lambda_i, \lambda_j$). Below we discuss how the models for spilling, I/O and CPU need to be adapted for each algorithm, for the general case where the input relations are seen as unions of subrelations. We assume that relations $R, S$ have respectively $n_R$ and $n_S$ subrelations.

The CPU cost model can directly utilize the above distributive property of relational algebra since it counts the number of comparisons for the join. That is, the total number of comparisons can be calculated by adding up the number of comparisons for each subrelation join. When it comes to the I/O cost model (and determining whether spilling occurs), if we consider the many subrelation joins separately, the model will not be able to correctly calculate the memory needs of the full join operation. Hence, for the I/O cost model for complex datasets we focus on the worst case scenario. The I/O model thus picks the duration and arrival rate from the subrelations that would create the worst case memory requirements.

**Calculating the CPU Cost.** The CPU cost model for multiple relations utilizes the union of $n$ individual subrelations to estimate the CPU cost. The cost model sums up the costs of all subrelation joins' CPU cost to create a complex dataset model.

$$C_{CPU.complex}(R, S) = \sum_{i=0}^{n_R} \sum_{j=0}^{n_S} C_{CPU}(R.i, S.j) \tag{4.66}$$

The TS CPU cost model has a separate formula for CPU when the algorithms spills. Since the I/O cost model generally use the worst case, the CPU unproductive comparisons for the TS formula also uses the worst case, instead of the union of all sub relations.

$$C_{CPU.complex.spill}(R, S) = \sum_{i=0}^{n_R} \sum_{j=0}^{n_S} C_{CPU}(R.i, S.j) + C_{CPU.spill}(R, S) \tag{4.67}$$

**I/O Spilling Check.** Consider the complex dataset and its' subrelations' interval properties. We will approximate the complex dataset with one duration and one arrival rate, those that will create the worst case I/O. First, we note the most I/O is caused when processing the longest duration. Thus, the largest duration from all subrelations is selected for the worst case. That is: $d_R = \underset{1 \leq i \leq n_R}{max}(d_{R.i})$. Second, the worst case for the interval arrival rate occurs at a time instance with the most arriving intervals. Since each subrelation may contribute to the number of tuples arriving on a specific time instance, the cumulative arrival rate is selected. That is: $\lambda_R = \sum_{i=1}^{n_R} \lambda_{R.i}$.

The SM algorithm stores data in memory according to a single tuple $t_R$ selected from $R$. The worst case scenario happens for the tuple with the longest interval in $R$, since it overlaps with the largest number of $S$ tuples (considering the matches of the longest $t_R$ with tuples from every $S_j$), and these need to be stored in memory. If all the matching $S$ tuples cannot be stored in memory the algorithm will spill. The original SM spilling formula (Formula 4.5) is replaced by the following formula:

$$\sum_{j=1}^{n_S} \frac{\lambda_{S.j} * \left( \underset{1 \leq i \leq n_R}{max}(d_{R.i}) + d_{S.j} - 1 \right)}{T_{f.S}} > F_M - 4 \tag{4.68}$$

The Time-Sweep Interval Join keeps all active tuples in memory to perform the join. In this case, we need to add up the active tuples in each subrelation (namely $L_{R.i}, L_{S.j}$), from both inputs. Hence Formula 4.20 is replaced by:

$$\sum_{i=1}^{n_R} \frac{L_{R.i}}{T_{f.R}} + \sum_{j=1}^{n_S} \frac{L_{S.j}}{T_{f.S}} > F_M - 5 \tag{4.69}$$

The Forward Scan Interval Join keeps tuples in memory from both datasets. For the longest tuple from dataset $S$ we want to find all $R$ tuples that overlap with it (arriving with rate $\lambda_i$ from each $R_i$); similarly for the longest tuple from $R$. Hence the spilling Formula 4.34 becomes:

$$\frac{\max\limits_{1 \le k \le n_S} (d_{S.k}) \sum_{i=1}^{n_R} \lambda_{R.i}}{T_{f_R}} + \frac{\max\limits_{1 \le l \le n_S} (d_{R.l}) \sum_{j=1}^{n_S} \lambda_{S.j}}{T_{f_S}} > F_M - 5 \tag{4.70}$$

Note that the spilling formulas for the partition-based algorithms (namely, OIP's Formula 4.48 and DIP's Formula 4.59), remain unchanged since for these algorithms, the spilling test is based on dataset size only.

**Calculating the I/O Cost.** The formulas for the I/O cost model for SM, TS, and FS are the same, except for replacing the worst case duration and arrival rate.

The OIP algorithm is able to handle the individual subrelations since it uses nested loop join (and thus the distributive property holds). The I/O cost model is split into write and read formulas. The write formula is only called once since we write the whole relation to disk. The read cost model can sum up the I/O for each subrelation pair being joined together. Hence the I/O cost model formula is:

$$C_{IO.complex}(R, S) = C_{IO.write}(R, S) + \sum_{i=0}^{n_R} \sum_{j=0}^{n_S} C_{IO.read}(R.i, S.j) \tag{4.71}$$

The DIP algorithm chooses partitions based on the number of active intervals in Formula 4.60. The $I_{total}(R)$ formula must be updated to consider the number of active intervals from all relations. The modified formula is:

$$I_{total}(R) = \sum_{i=1}^{n_R} L_{R.i} \tag{4.72}$$

## 4.5 Model Accuracy

The accuracy for predicting the number of comparisons (CPU) and the number of disk accesses (I/O) was tested by logging the number of join comparisons and disk accesses while executing interval join queries. During this process, the cost model highlighted a few issues with the implementation by showing where the algorithm implementation did not match the model, usually finding a coding mistake or opportunity for improvement. A few of the algorithm implementations had an extra comparison or disk access often due to using a condition which included more tuples than necessary. The model accuracy experiments have been split into two sections: simple dataset and complex dataset. In all experiments the relation is self-joined. Each section shows the results using various duration and $\lambda$ values for both the CPU cost model and I/O cost model accuracy using synthetic data.

### 4.5.1 Simple Dataset Self-Join

The dataset default properties are $\lambda = 10$ and $d = 10$ unless stated otherwise.

**CPU Cost Model Accuracy.** The cost model accuracy test uses the same synthetic data generator from the performance testing in Section 3.4. While the interval data properties for each relation have been varied for each accuracy test, the same performance interval join queries have been used for these five algorithms. Table 4.4 shows the percent error ($|\frac{v - v_{estimate}}{v}| * 100\%$ where $v$ is the observed number of comparisons) using the simple

| **Duration** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Duration of 1 | 0.01% | 0.001% | 1.81% | 0.00% | 0.00% |
| Duration of 5 | 0.01% | 1.12% | 2.33% | 1.11% | 0.01% |
| Duration of 10 | 0.01% | 0.57% | 2.78% | 0.53% | 0.01% |
| Duration of 50 | 0.01% | 0.35% | 7.57% | 0.11% | 0.01% |
| Duration of 100 | 0.01% | 0.55% | 11.79% | 0.06% | 0.01% |

Table 4.4: CPU Cost Model error for a dataset with a $\lambda = 10$ and duration is shown.

model, where intervals arrive with the same arrival rate; their duration $d$ varies from 1 to 100 time instants. The CPU Cost Model Accuracy builds on the join size estimation accuracy from Section 4.3. Many of the CPU cost models use the join size estimation formula and add the unproductive comparison to create the cost model.

Overall, the errors are typically below 3% with the exception of a few OIP scenarios. DIP and TS show consistent low error values. DIP has unproductive comparisons but are predictable and accounted for accurately. TS does not have unproductive comparisons, so the accuracy is based on the join size estimation. OIP's error comes from assuming all partitions have the same size and number of tuples. If the relation's range is not evenly divisible by the $k$ value, that implies that some partitions will be of different size and have an alternate number of tuples. The error size is related to many of these alternate size partitions exist. Table 4.5 shows the percent error from various lambda queries. The results are similar (the error is less than 5.04%) to the ones varying the duration.

| **Lambda** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Lambda of 1 | 0.01% | 5.04% | 1.90% | 5.01% | 0.01% |
| Lambda of 5 | 0.01% | 1.09% | 0.72% | 1.05% | 0.01% |
| Lambda of 10 | 0.01% | 0.57% | 2.78% | 0.53% | 0.01% |
| Lambda of 50 | 0.01% | 0.15% | 3.31% | 0.11% | 0.01% |
| Lambda of 100 | 0.01% | 0.10% | 0.47% | 0.06% | 0.01% |

Table 4.5: CPU Cost Model error for a dataset with a $d = 10$ and lambda is shown.

| **Duration** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Duration of 50 | 1.65% | 2.28% | 0.31% | 3.08% | 1.77% |
| Duration of 100 | 1.71% | 0.99% | 1.00% | 1.81% | 3.28% |

Table 4.6: I/O Cost Model error for a dataset with a fixed $\lambda = 10$ and two durations.

**I/O Cost Model Accuracy** The interval joins for the I/O cost models follow the performance I/O queries from Section 3.4.3. The experimental settings were selected to ensure all the available join memory is used and the algorithm must spill for all cases. The accuracy is only shown for the two longer intervals and two larger arrival rates since these were the only settings where all five interval joins spilled. The results are shown in Tables 4.6 and 4.7. Again OIP shows larger error (for $\lambda = 100$) since the simplified model assumes all partitions have the same size and number of tuples.

| **Lambda** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Lambda of 50 | 0.48% | 7.93% | 2.61% | 0.51% | 2.79% |
| Lambda of 100 | 0.49% | 3.06% | 28.82% | 0.32% | 7.69% |

Table 4.7: I/O Cost Model error for a dataset with a fixed $d = 10$ and two arrival rates.

### 4.5.2 Complex Dataset Self-Join

To create a complex dataset we add a second subrelation (with different $d$ and $\lambda$) to the relation that is self-joined. Thus the relation now has two types of intervals: a fixed sub-relation with a $\lambda_1 = 5$ and $d_1 = 5$ and a second sub-relation whose $\lambda$ and $d$ change for each experiment. In the duration experiments, the second subrelation uses $\lambda_2 = 10$ and varies $d_2$, while in the arrival rate experiments, the second subrelation uses $d_2 = 10$ and varies $\lambda_2$.

**CPU Cost Model Accuracy** Table 4.8 shows the model accuracy results for the complex dataset case, where the duration of the second subrelation is changed. While the error of the complex dataset self-join is higher that the simple dataset case, the numbers are still below 4% error except for a two OIP experiments. The accuracy results while changing the second subrelation $\lambda$ appear in Table 4.9. Again the error is low, showing the accuracy of our models holds for the complex case as well.

**I/O Cost Model Accuracy** The I/O cost model accuracy tests for the complex dataset appear in Table 4.10 (while changing the second subrelation's duration), and in Table 4.11 (changing the second subrelation's arrival rate). Excluding TS and OIP, the errors for the duration experiments are below 3%. In the arrival rate experiments the error

| **Duration** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Duration of 1 | 2.33% | 3.53% | 3.48% | 1.20% | 3.52% |
| Duration of 5 | 1.11% | 1.48% | 2.46% | 1.47% | 1.47% |
| Duration of 10 | 0.67% | 3.33% | 2.09% | 1.26% | 0.85% |
| Duration of 50 | 0.11% | 0.53% | 5.20% | 0.28% | 0.20% |
| Duration of 100 | 0.05% | 0.09% | 9.37% | 0.14% | 0.11% |

Table 4.8: CPU cost model error for a dataset with two types of interval data: $\lambda_1 = 5$, $d_1 = 5$, $\lambda_2 = 10$ and various $d_2$ values.

| **Lambda** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Lambda of 1 | 3.68% | 3.06% | 1.42% | 3.04% | 3.04% |
| Lambda of 5 | 0.01% | 5.59% | 3.41% | 2.10% | 1.42% |
| Lambda of 10 | 0.67% | 3.33% | 2.09% | 1.26% | 0.85% |
| Lambda of 50 | 3.40% | 0.76% | 1.44% | 0.29% | 0.21% |
| Lambda of 100 | 4.04% | 0.36% | 3.34% | 0.14% | 0.11% |

Table 4.9: CPU Cost Model error for a dataset with two types of interval data: $\lambda_1 = 5$, $d_1 = 5$, $d_2 = 10$ and various $\lambda_2$ values.

| **Duration** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Duration of 50 | 0.42% | 1.26% | 0.35% | 1.21% | 16.33% |
| Duration of 100 | 6.98% | 1.93% | 2.58% | 1.00% | 18.78% |

Table 4.10: I/O Cost Model error for a dataset with two types of interval data: $\lambda_1 = 5$, $d_1 = 5$, $\lambda_2 = 10$ and two $d_2$ durations.

| **Lambda** | **DIP** | **FS** | **OIP** | **SM** | **TS** |
|---|---|---|---|---|---|
| Lambda of 50 | 0.74% | 7.15% | 46.49% | 2.23% | 4.51% |
| Lambda of 100 | 0.07% | 3.80% | 29.88% | 0.03% | 6.68% |

Table 4.11: I/O Cost Model error for a dataset with two types of interval data: $\lambda_1 = 5$, $d_1 = 5$, $d_2 = 10$ and two $\lambda_2$ values.

is higher, but generally less then 8% except for the OIP results. OIP continues to have a higher error rate for large arrival rates for I/O experiments similar to the simple dataset case.

## 4.6 Predicting Spilling

When picking an algorithm for the best performance, an optimizer should pick an algorithm that avoids spilling due to the significant increase in execution time. The join memory estimate may be used to identify which algorithm will spill to disk. In Figures 4.12 and 4.13, two graphs are shown that detail when different algorithms will spill based on the interval data. Note that the partitioning algorithms spill based on the size of the data and were thus left out of the figures. The non-partitioning algorithms spill based on the interval data properties. Consider two datasets $R$ and $S$. In Figure 4.12, dataset $R$ has a fixed

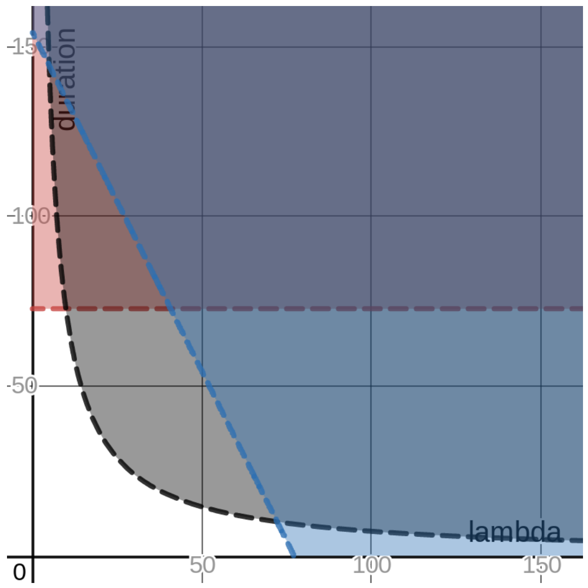


Figure 4.12: The graph's x and y coordinates represent $\lambda_S$ and $d_S$ for the $S$ dataset and the $R$ dataset is fixed with $d_R = 5$ and $\lambda_R = 10$. The shaded regions represent when algorithms will spill: SM is red, TS is black, and FS is blue.

arrival rate $\lambda_R = 10$ and a fixed duration $d_R = 5$. Dataset $S$'s arrival rate $\lambda$ is shown on the X axis while the interval duration for $S$ is on the Y axis. The shaded regions depict when an algorithm will spill during the join. The red region represents when the SM algorithm, the black region represents TS algorithm and blue region represents the FS algorithm.

The figure represents how the spilling formulas can be used to pick an algorithm that does not spill. As an example, if dataset $S$ had $\lambda_S = 40$ and $d_S = 40$, it would correspond to a point in the black region, which means that TS would spill for this dataset, while SM or FS would not spill.

In Figure 4.13 we depict the spilling scenarios for a dataset $R$ with duration $d_R = 15$, while keeping a fixed arrival rate $\lambda_R = 10$. These graphs are hosted on Demos and can be visualized at the following URL in the references [31].

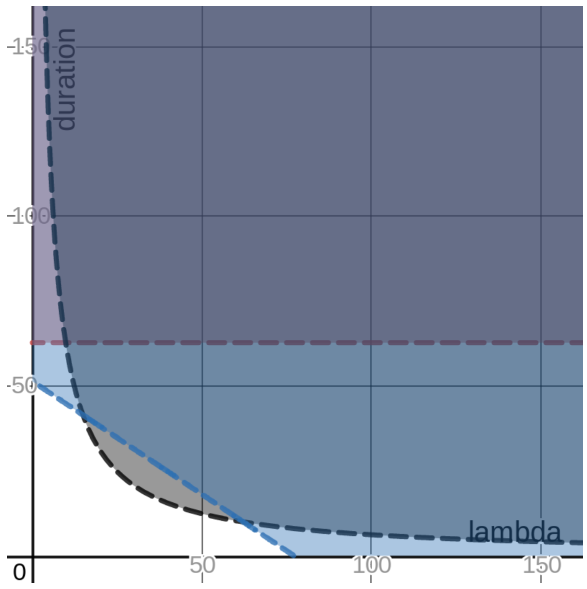


Figure 4.13: The graph's x and y coordinates represent $\lambda_S$ and $d_S$ for the $S$ dataset and the $R$ dataset is fixed with $d_R = 15$ and $\lambda_R = 10$. The shaded regions represent when algorithms will spill: SM is red, TS is black, and FS is blue.

## 4.7 Conclusions

We presented a interval join size estimation formula that accurately predicts the number of results with an error of less than one percent. The CPU cost models for the five interval algorithms show relatively good accuracy for various durations and arrival rates. The I/O cost model is not as accurate because the model is based on worst case scenarios. Overall, the spilling formulas can be used to pick an algorithm that avoids spilling, as this drastically affects performance. These cost models became a valuable check point when implementing the various join algorithms. Often an issue with the implementation was highlighted by the cost model giving a wildly different value. These updates helped optimized and confirm the implementation by comparing them to the cost model. The cost models can be integrated into a cost-based query optimizer when integrating these an interval join into a database system. Since spilling is costly for a query, the primary goal should be to use the cost models to avoid spilling.

Chapter 5

# Lessons Learned From Implementing the Overlapping Interval Join Algorithms

There were various lessons learned from implementing, building cost models and testing the five interval join algorithms (FS, SM, TS, OIP and DIP) for the overlapping relationship. The non-partition algorithms (FS, SM, and TS) have similar spill processes and only differ slightly in the way they manage tuples in memory during the join process. This is also confirmed by the cost models for the non-partition algorithms which predict close CPU and I/O costs. The SM algorithm was simple to implement and performs well in many scenarios. SM is the only non-partitioning algorithm that is designed to only spill one dataset. The FS and TS algorithms may spill both datasets depending on the input data. Note these non-partitioning algorithms share the same spill process for freeing a dataset from memory. The FS and TS implementation uses an updated memory manager in AsterixDB that can process two datasets in memory by adding/removing tuples from both. The TS algorithm's approach of keeping active tuples in memory made it the top performer in many of the experiments due to the reduced number of CPU comparisons.

The partitioning algorithms' approach for determining matching tuples involves pre-partitioning each dataset which requires some amount of I/O during processing each query. A disadvantage of the DIP algorithm it that it may create large number files so as to manage all the algorithm's partitions. This may require adjusting the number of open files that a system can handle if one decides to implement this algorithm. As the join memory decreases, the DIP algorithm is the first to show performance degradation due to early spilling. The DIP cost model confirms that the algorithm uses significantly higher I/O for similar join queries when compared to the non-partitioning algorithms. As for the other partitioning approach, OIP, we found that its performance is connected to selecting a good number of slots ($k$ value). The formula to calculate $k$ depends on the algorithm's input settings based on the given CPU speed and disk I/O speed.

In an effort to summarize the behavior of each algorithm, we compiled all the experiments from Section 3.4 into Figure 5.1 and (its detailed version) Figure 5.2. All 63 experiments have been compiled into these figures. The x-axis represents the experiments, numbered from 1 to 63, following the same order as they appear in Section 3.4. That is, the first experiment comes from Figure 3.18 from the "Count Only" query, followed by the second experiment which corresponds to the "Empty Result" from the same figure, and so on. Each of the five algorithms is represented by a different dot. The position of a

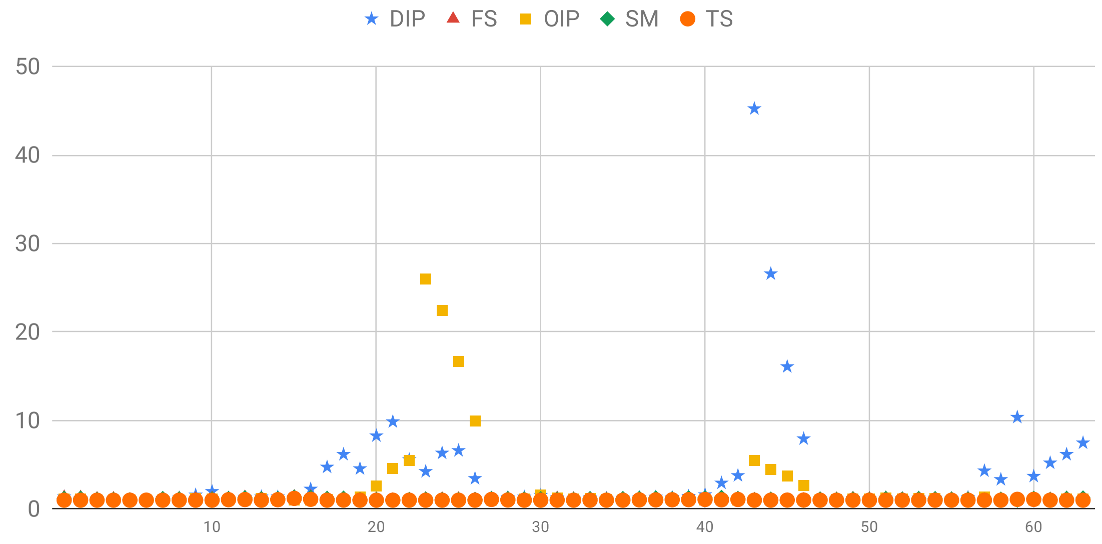


Figure 5.1: The overlapping join experiments where each algorithm's performance is depicted as a ratio to the performance of the fastest algorithm for that query.

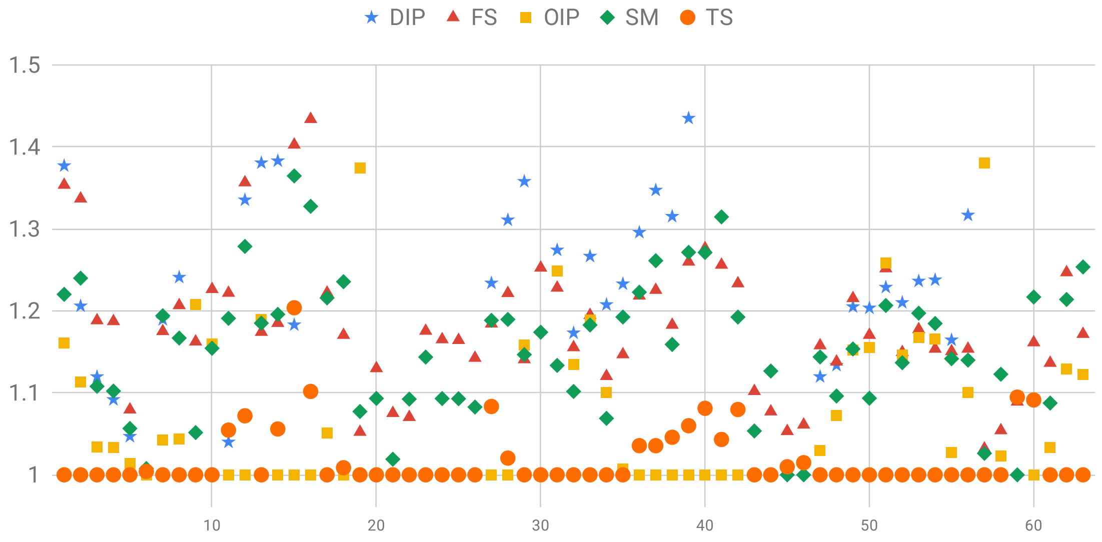


Figure 5.2: The overlapping join experiments of Figure 5.1 focusing on ratios between 1 and 1.5.

particular dot (algorithm) on the y-axis represents this algorithm's performance as a ratio to the fastest query time in that performance. Here the dot with value 1 corresponds to the algorithm with the fastest performance; an a dot with value 1.25 corresponds to an algorithm that was 25% slower than the fastest for that experiment.

Figure 5.1 shows that the DIP and OIP algorithms were much slower in several experiments (in many cases between 5 and 45 times slower). To better visualize the performance of the best performing algorithms, Figure 5.2 zooms on the algorithms that were up to at most 1.5 slower than the fastest algorithm in the particular experiment (i.e., up to 50% slower).

Overall, the TS algorithm was the fastest in 43 of the experiments, followed by OIP which was the fastest in 17 and SM in 3. Based on its robust performance the TS algorithm would be a top choice for implementing a interval join algorithm. Even when it is not the best performing algorithm, its performance was consistently at most 10% worse than the best algorithm (except in one experiment where it was 20% worse). The performance of SM was no worse than 26% slower from TS and the performance of FS was no worse than 35% slower than TS. SM is an easy algorithm to implement and it may take advantage of an existing interval order. The second top performer was OIP in several of the experiments. Yet as the record size increased, the algorithm's performance was noticeably slower than the non-partitioning algorithms due to spilling. In 11 experiments the OIP was more than two times slower than the fastest algorithm.

Another important performance characteristic is that when spilling occurs, all algorithms slow down significantly (when compared with the non-spilling case performance) as their speed is now dependent on I/O. Figure 5.3 shows the last seven of the experiments (experiment 57 to 63 from the previous Figure) where all algorithms spill. Experiment 57 corresponds to the case with record of size 2,222 bytes from Figure 3.31; all experiments after that have large record size so all algorithms spill. In particular, when the DIP spills, it has a significantly higher I/O than the rest. Among the rest, the performance is within 25% of the best algorithm (except for one case where OIP is 38% worse). This can be observed in the detailed Figure 5.4 that shows the same experiments in the performance ratio between 1 and 1.5. As a result, the first choice for an optimizer is to select an algorithm that avoids spilling (see Section 3.31). If there are many candidate algorithms that do not spill, the optimizer can use the CPU-based cost model to choose the best non-spilling algorithm.

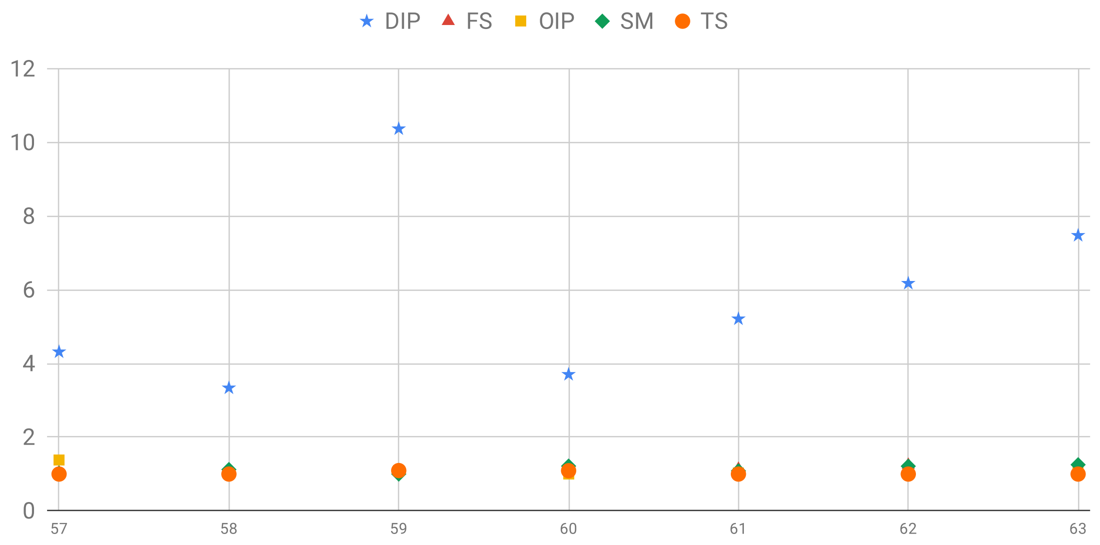


Figure 5.3: The spill overlapping join experiments where each algorithm's performance is depicted as a ratio to the performance of the fastest algorithm for that query.

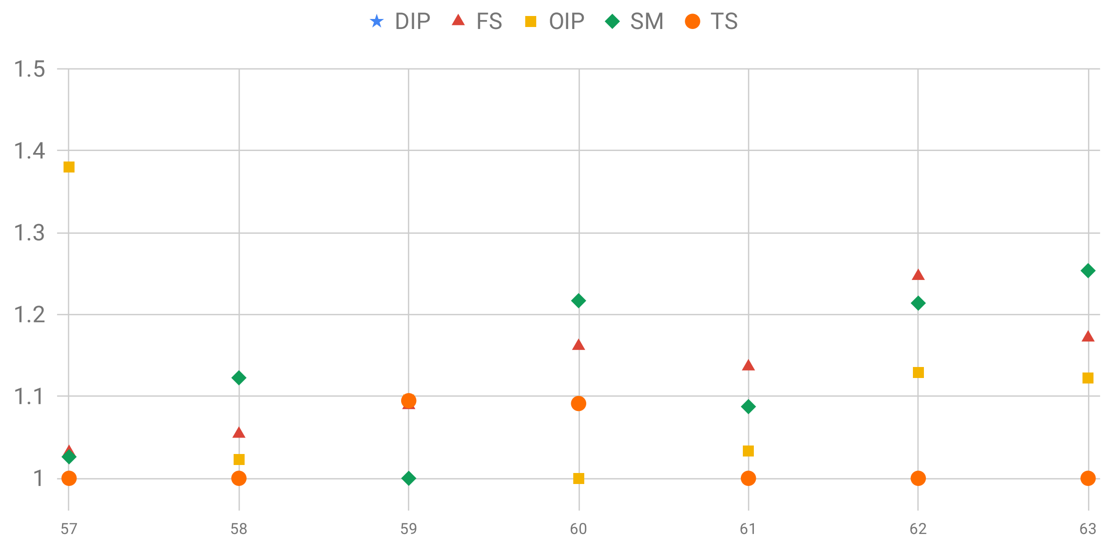


Figure 5.4: The spill overlapping join experiments of Figure 5.3 focusing on ratios between 1 and 1.5.

Chapter 6

# Scaling Joins Involving Allen's Interval Algebra

## 6.1 Introduction

Most previous work on interval joins have focused on the overlapping relationship (i.e. if two intervals share a common part), since it is very generic. Nevertheless, Allen's algebra defines thirteen interval relationships [2] that describe all possible ways one interval can relate to another, using the start and end points of the two intervals. The formula that describes overlapping is shown in Table 6.1 while the formulas describing Allen's relationships are shown in Table 6.2, respectively. Note that we use the AsterixDB join terms. That is "ends," corresponds to Allen's "finishes"; "ended-by" corresponds to "finished-by"; "covers" replaces "during"; and "covered-by" is used in place of "contains".

| **Interval Join** | **Formula** |
|---|---|
| overlapping($t_R$,$t_S$) | $(t_R.start <= t_S.start \,\&\&\, t_R.end > t_S.start)$ $\|\| \, (t_R.end >= t_S.end \,\&\&\, t_R.start > t_S.end)$ |

Table 6.1: Overlapping interval formula.

| **Interval Join** | **Formula** |
|---|---|
| before($t_R$,$t_S$), after($t_S$,$t_R$) | $t_R.end < t_S.start$ |
| ends($t_R$,$t_S$), ended-by($t_S$,$t_R$) | $t_R.end = t_S.end \,\&\&\, t_R.start >= t_S.start$ |
| equals($t_R$,$t_S$) | $t_R.start = t_S.start \,\&\&\, t_R.end = t_S.end$ |
| meets($t_R$,$t_S$), met-by($t_S$,$t_R$) | $t_R.end = t_S.start$ |
| starts($t_R$,$t_S$), started-by($t_S$,$t_R$) | $t_R.start = t_S.start \,\&\&\, t_R.end <= t_S.end$ |
| covers($t_R$,$t_S$), covered-by($t_S$,$t_R$) | $t_R.start <= t_S.start \,\&\&\, t_R.end >= t_S.end$ |
| overlaps($t_R$,$t_S$), overlapped-by($t_S$,$t_R$) | $t_R.start < t_S.start \,\&\&\, t_R.end < t_S.end$ $\&\&\, t_R.end > t_S.start$ |

Table 6.2: Formulas for Allen's interval algebra (using AsterixDB join terms).

While each of the formula's in Table 6.2 detail a join condition for an interval join, these conditions do not always require a special interval join operator. For example, the first two relationships, *before* and *after*, can each be addressed as a traditional relational inequality join, which can be optimized by appropriate partitioning (based on the start or end of the relation intervals). [20] actually creates a new partitioning strategy just for this type of join to reduce the long running time of processing the last partition. While these joins may be time consuming, their unique features are not specifically related to intervals.

The next seven relationships (*ends*, *ended-by*, *equals*, *meets*, *met-by*, *starts* and *started-by*) include at least one condition with an equality. Thus, they can all be addressed by existing (and very efficient) equi-join algorithms (with a post filter for applying any other join condition). For example, the *starts* interval join could use hash partitioning on the $t_R.start = t_S.start$ condition and then later filter the join results for the inequality part of the condition ($t_R.end <= t_S.end$). The hash partitioning would ensure intervals with the same start time instance are grouped in the same partition (using the time points from the equal condition). Using the hash partition operator would mean the dataset does not need to be sorted.

The remaining relationships, *covers*, *covered-by*, *overlaps* and *overlapped-by* search for shared parts between the two intervals and are thus closer to the *overlapping* relationship we have examined. In this chapter we look at the five overlapping interval algorithms (previously discussed in Chapter 3) and show how they can be modified to support each of these four Allen's join conditions. Note that an algorithm that can solve the *covers* join condition can be used (through simple rewriting) to also solve the *covered-by* join condition.

Similarly, for the *overlaps* and *overlapped-by* join conditions. This is comparable to how current systems implement left-outer-join by the same algorithm used for the right-outer-join, by changing the order of the relations. As a result, in Section 6.2 we concentrate on the *covers* and *overlaps* join conditions. In particular, we first detail how the SM algorithm can be modified to answer *covers* and *overlaps* and then highlight the changes needed for the other interval join algorithms to address them. Section 6.3 presents performance results for *covers* and *overlaps* joins using the modified SM algorithm, following a similar set of experiments as in Chapter 3. Finally, the Section 6.4 concludes this chapter.

## 6.2 *Covers* and *Overlaps* Interval Join Algorithms

*Covers* and *Overlaps* interval joins deal with joining intervals over a range of values similar to an overlapping interval join. An overlapping interval join includes all the results for *overlaps*, *overlapped-by*, *covers*, and *covered-by*. A simple implementation would be to execute an overlapping interval join and then filter for the alternate interval join predicate for Allen's relation: *overlaps* or *covers*. While using the overlapping interval join algorithms would get the correct answer, many extra interval comparisons would be made for pairs that are not part of the result. In addition, these intervals would take up space in memory, thus decreasing the efficiency and increasing the chance of spilling during the join. Since the Allen's intervals all have tighter requirements, the algorithms may be able to use memory more efficiently and reduce unnecessary join comparisons.

The query plan must choose an efficient global partitioning strategy for Allen's relations. Fortunately [20] has provided a global partitioning strategy for Allen's relations shown

| **Interval Join** | **Inverse** | **Global Partitioning Schemes** |
|---|---|---|
| $Before(t_R, t_S)$ | $After(t_S, t_R)$ | *Replicate*($t_R$) & *Project*($t_S$) |
| $Ends(t_R, t_S)$ | $EndedBy(t_S, t_R)$ | *Project*($t_R$) & *Project*($t_S$) |
| $Equals(t_R, t_S)$ | | *Project*($t_R$) & *Project*($t_S$) |
| $Meets(t_R, t_S)$ | $MetBy(t_S, t_R)$ | *Project*($t_R$) & *Project*($t_S$) |
| $Starts(t_R, t_S)$ | $StartedBy(t_S, t_R)$ | *Project*($t_R$) & *Project*($t_S$) |
| $Covers(t_R, t_S)$ | $CoveredBy(t_S, t_R)$ | *Split*($t_R$) & *Project*($t_S$) |
| $Overlaps(t_R, t_S)$ | $OverlappedBy(t_S, t_R)$ | *Split*($t_R$) & *Project*($t_S$) |

Table 6.3: Global join partitioning strategies for Allen's interval algebra joins for datasets $R$ and $S$ (taken from [20]).

in Table 6.3. While the overlapping interval join (from section 3.2) required both datasets to be globally *Split* partitioned (that is, *Split*($t_R$) & *Split*($t_S$)), *overlaps*, *overlapped-by*, *covers*, and *covered-by* all use *Split* partitioning for each tuple $t_R$ from relation $R$ and *Project* partitioning for each tuple $t_S$ from relation $S$ (i.e., *Split*($t_R$) & *Project*($t_S$)). *Project* partitioning corresponds to range partitioning from Section 2.4, using the interval's start point as the partitioning key. In comparison with the *Split* partitioning, no intervals are replicated in the *Project* partitioning scheme, thus there is less network communication and fewer intervals in each partition.

Using these interval global partitioning schemes, the interval join algorithms focus on managing memory by keeping only interval tuples which can possibly match with other intervals. An algorithm's conditions for adding and removing the intervals from memory may need to be updated for these new interval relationships. The new condition for man-

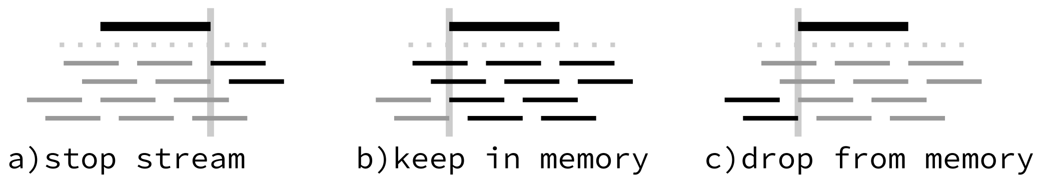


Figure 6.1: Overlapping intervals: a) stop processing stream, b) keep in memory for future stream tuples, and c) may remove from memory since no future stream tuples will be joined.

aging memory should minimize the unnecessary interval join checks for the updated query predicates. The next section will take a deep look at modifying the Sort-Merge Interval Join and then a brief look at how to apply similar techniques to the other interval join algorithms.

### 6.2.1 Base Case: Sort-Merge Interval Join

Recall that in the Sort-Merge Interval Join (SM) algorithm, $t_R$ tuple is selected from the Dataset $R$ and then joined with all possible tuples from Dataset $S$. Since the input source is a stream of data, tuples from $S$ must be stored in memory so that future tuples from $R$ may be joined. Three scenarios are important to consider for managing these two input streams and the available join memory. First, the time instance when the tuple $t_R$ will no longer match with any future tuple from $S$. This condition identifies when the join can move on to the next tuple $t'_R$ in the stream of $R$. The next two conditions focus on managing when a tuple from $S$ is added or removed from memory. When a tuple $t_S$ is processed from the stream, the second condition checks to see if the tuple needs to be kept in memory for future matches. The final condition identifies when an in-memory interval $t_S$ will no longer match with any future tuples from the stream of $R$.

The three conditions are highlighted in the following Figures 6.1 - 6.5. In all figures, the bold black line above the dotted line represents the active stream interval $t_R$. The lines below the dotted line represent the stream of tuples from Dataset $S$. The stream data is ordered by start point followed by end point for each tuple. The vertical line represents the condition in the three Figures.

First, the overlapping condition is shown in Figure 6.1. In particular, the vertical line in Figure 6.1a shows when intervals from Dataset $S$ no longer match with the selected interval from Dataset $R$. The black lines under the dotted line show which tuples start after the vertical line and are not overlapping the selected interval from Dataset $R$. Figure 6.1b shows all the tuples from $S$ that could match with future tuple from the stream of Dataset $R$. Future tuples will always start at the same time or later than the selected tuple. Figure 6.1c shows all the tuples that could be removed from memory since any future tuple from the stream of Dataset $R$ will not be overlapping.

Figures 6.2, 6.3, 6.4, and 6.5 demonstrate the same three conditions for the rest of the joins (including, for reference, overlapped-by and covered-by). Sub-figure (a) does not change for the five interval joins. The primary difference comes for sub-figures (b) and (c). For example, Figure 6.2b and 6.4b do not include intervals from $S$ that start before $t_R$. Figure 6.2c and 6.4c do include intervals from $S$ that start before $t_R$ (instead of strictly end before $t_R$ as in overlapping). This means fewer tuples need to be stored in memory for *overlaps* and *covers*. Figures 6.3 and 6.5 show the same properties but with Dataset $R$ and $S$ switched.

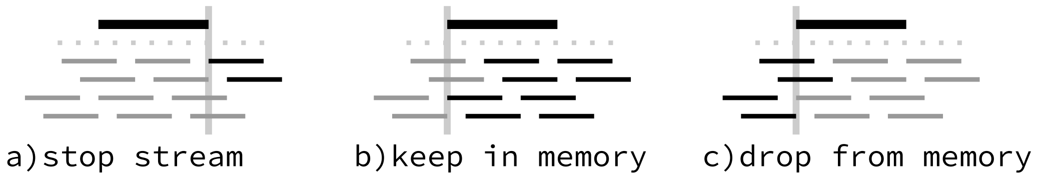


Figure 6.2: *Overlaps* intervals: a) stop processing stream, b) keep in memory for future stream tuples, and c) may remove from memory since no future stream tuples will be joined.

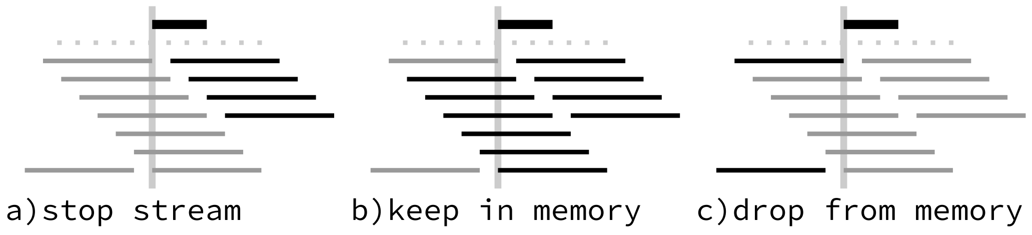


Figure 6.3: *Overlapped By* intervals: a) stop processing stream, b) keep in memory for future stream tuples, and c) may remove from memory since no future stream tuples will be joined.

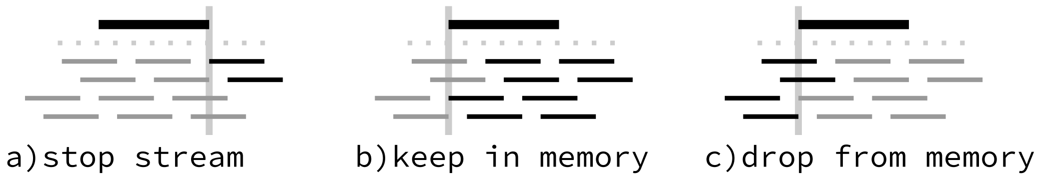


Figure 6.4: *Covers* intervals: a) stop processing stream, b) keep in memory for future stream tuples, and c) may remove from memory since no future stream tuples will be joined.

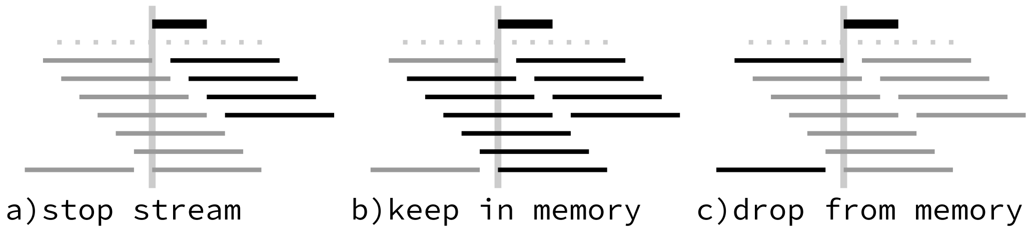


Figure 6.5: *Covered By* intervals: a) stop processing stream, b) keep in memory for future stream tuples, and c) may remove from memory since no future stream tuples will be joined.

| **Interval Join** | **More Matches** | **Add To Memory** | **Drop From Memory** |
|---|---|---|---|
| Overlapping | $t_R.end > t_S.start$ | $t_R.start < t_S.end$ | $t_R.start >= t_S.end$ |
| Overlaps | $t_R.end > t_S.start$ | $t_R.start <= t_S.start$ | $t_R.start > t_S.start$ |
| Overlapped-By | $t_R.start <= t_S.start$ | $t_R.start <= t_S.end$ | $t_R.start > t_S.end$ |
| Covers | $t_R.end > t_S.start$ | $t_R.start <= t_S.start$ | $t_R.start > t_S.start$ |
| Covered-By | $t_R.start <= t_S.start$ | $t_R.start <= t_S.end$ | $t_R.start > t_S.end$ |

Table 6.4: Conditions for managing join memory of $R$ and $S$.

Figures 6.1 - 6.5 help us identify the changes in the SM algorithm memory management (during the local join); these changes are depicted in Table 6.4. Interestingly, the conditions for the four Allen's relations are similar (if not the same) to the conditions for overlapping (also shown in the Table). The differences are based on the tighter requirements that Allen's interval algebra has with the interval join definition.

### 6.2.2 Modifying the Other Join Algorithms

The Time-Sweep Interval Join keeps all active tuples at a single point in time in memory. Since this is unrelated to the join predicate implemented, no changes are made to the algorithm except for which join predicate to check. However the algorithm's memory management can be improved by only storing one dataset in memory instead of two. Consider the $covers(t_R, t_S)$ join condition; this condition is not symmetric. That is, if a tuple $t_R$ *covers* a tuple $t_S$ the opposite may not be true. This is in contrast to the *overlapping* relation, where if a tuple $t_R$ is overlapping tuple $t_S$, then $t_S$ is *overlapping* with $t_R$ as well, and thus the algorithm benefits from storing tuples from both relations in memory. As a result, the TS algorithm needs only to load tuples from $R$ in memory and stream tuples from $S$. By keeping tuples from only one relation in memory decreases the chances that the algorithm will spill. The same applies also to the *overlaps* condition.

The Forward-Scan Interval Join selects the next time ordered tuple from either dataset and loads possible future matching tuples into memory. Similar to the TS algorithm, the join will produce results for all Allen's relations by only changing the join predicate. Moreover, the FS algorithm also only needs to store one dataset in memory instead of two for the same reason as TS.

The partitioning join algorithms require a different approach due to their alternate memory strategy. The Overlapping Interval Partition Join works by picking appropriate partitions to join. The join process itself does not need to change, just which partitions are chosen to be joined (which is implied by the Allen's relation considered).

The Disjoint Interval Partition Join creates partitions where the tuples in each partition do not overlap with tuples in that partition. As a result, it does not require any changes except for which predicate to implement. This implies that for DIP, the number of comparisons and disk accesses will be the same for all four Allen's relations.

## 6.3 Performance

We have conducted a set of performance experiments to look at how the updated SM algorithm performs in a real database system, AsterixDB, using the same cluster hardware and configuration. Three sets of experiments review different aspects of measuring the algorithm's performance: speed-up (when sufficient memory is available), scale-up (increasing data size and available resources), and scale-up with spilling (when limited memory requires the use of disk space). The following performance figures depict the join predicate *covers* and *overlaps*, (as well as *overlapping* for comparison purposes) for each test.

Note that a self-join using *covers* as the interval join predicate, will not produce any results since all intervals are of the same length. Thus dataset $R$ has been updated so that each interval has a duration twice the length of an interval in Dataset $S$. The rest of the data properties have been left the same. The specific interval data is detailed for each experiment below. Again queries report the average query time for the top 100 results.

### 6.3.1 Speed-Up Non-Spilling Experiments

We start with testing on a single node, followed by multi-node experiments.

**Single Node Speed-Up.** Figure 6.6 shows the modified SM algorithm on two synthetic datasets each with 10,000 records evenly distributed over a time range of 1,000 units ($\lambda = 10$). The tuples in Dataset $R$ have a fixed interval duration of $d_R = 20$, while the tuples in Dataset $S$ have a smaller duration of $d_S = 10$. The size of a tuple from each relation is 74 bytes. As before, the local node has four cores. In this experiment we start with one thread and one partition, and continue to double the threads and partitions. Because of hyper-threading, we can go up to 8 threads (and 8 partitions). All queries show good speed-up (except for when using hyper-threading which is expected since the node has only four cores). Note that *overlapping* takes consistently more time than *covers* and *overlaps*, due to the larger join result.

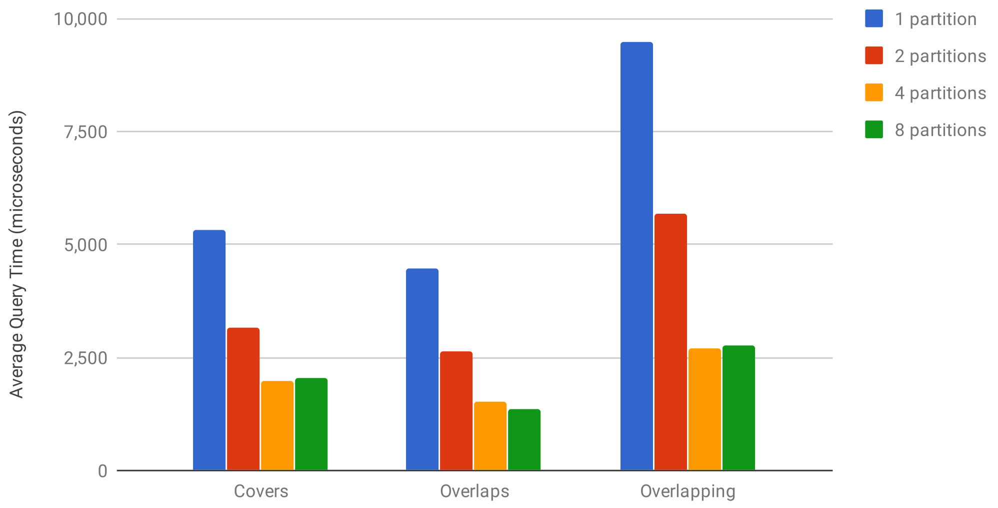


Figure 6.6: Single node speed-up performance of the modified SM algorithm on synthetic data ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 20$ and $d_S = 10$).

In Figure 6.7, the cardinality in both datasets has been increased ten times, while the time range and other properties have not been changed. As a result, there are ten times more tuples in relation $R$, and each such tuple will match with $S$ tuples that are ten times more. This results to 100 times more comparisons; as it can be seen in the figure the query time has increased similarly, when compared to the query time in Figure 6.6. Again, the modified SM algorithm shows good speed-up in all queries.

In Figure 6.8, the duration for both datasets has been increased ten times (that is, $d_R = 200$ and $d_S = 100$). As a result, each tuple in $R$ will now match with ten times as many $S$ tuples. Since the number of tuples has not changed (when compared with Figure 6.6) the number of comparisons increases by ten times. The modified SM algorithm shows again good speed up for all three queries.

Figure 6.9 examines the effect of large records on the modified SM algorithm's performance. The same interval relations as in Figure 6.6 are used but here the record size

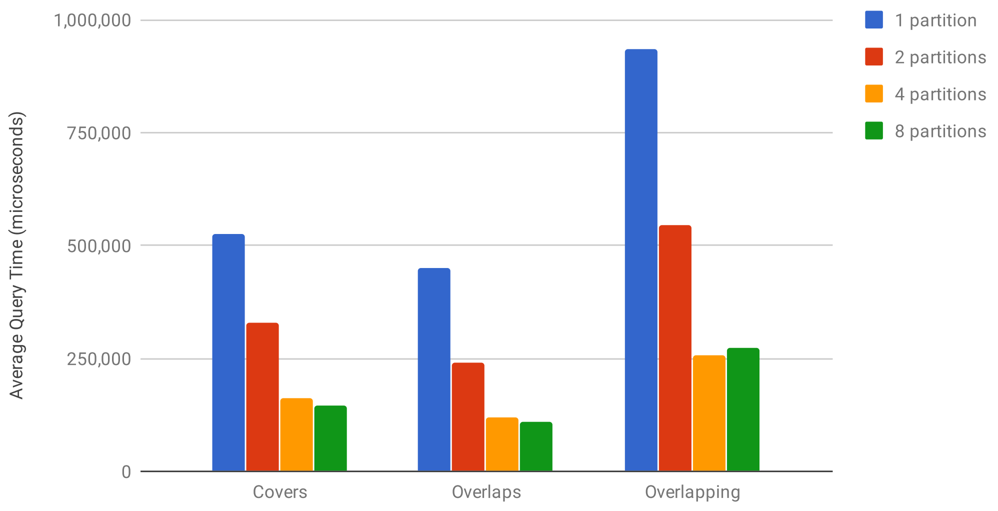


Figure 6.7: Single node speed-up performance of the modified SM algorithm on synthetic data ($\lambda_R = 100$, $\lambda_S = 100$, $d_R = 20$ and $d_S = 10$).

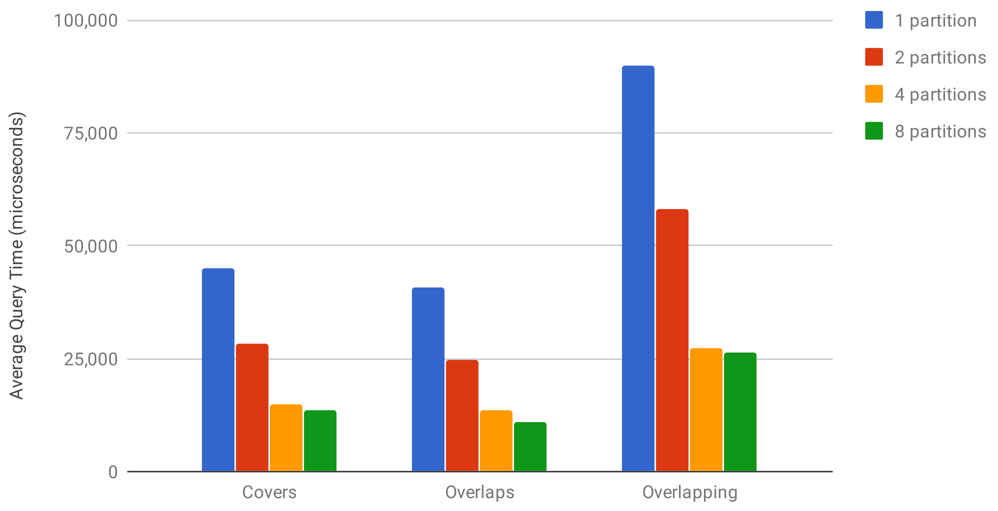


Figure 6.8: Single node speed-up performance of the modified SM algorithm on synthetic data ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 200$ and $d_S = 100$).

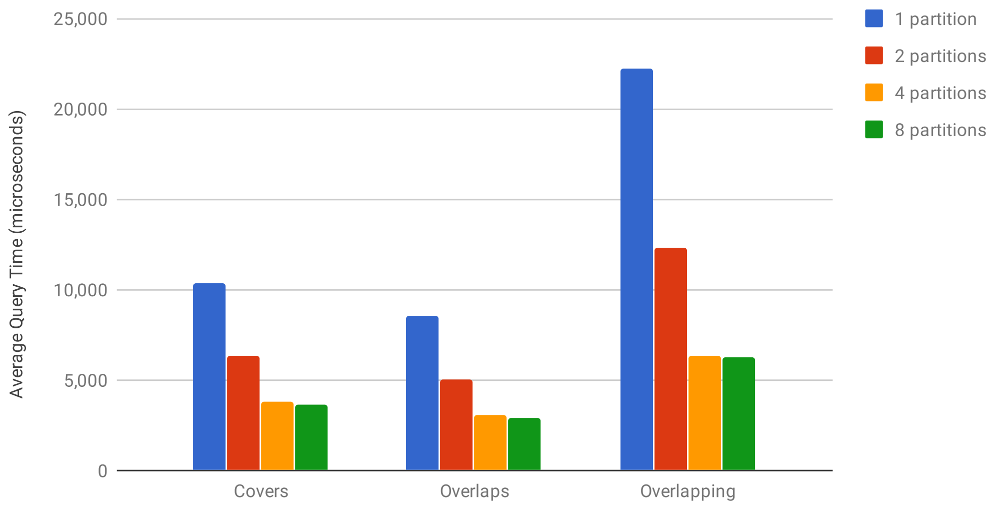


Figure 6.9: Single node speed-up performance of the modified SM algorithm on synthetic data with large records (2296 bytes) ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 20$ and $d_S = 10$).

in both relations has increased to 2296 bytes. The key difference is that the amount of data that is pushed through the join operator has increased thus resulting in higher query times as compared to Figure 6.6. The modified SM algorithm continues to show good speed-up for all three interval joins.

Finally, we experimented with a TPC-H generated dataset. (Note that the Infectious real dataset we considered in Section 3.4 will not produce results for the *covers* query since all its intervals have the same 20sec length). As before, the TPC-H generated data has durations of varying length (from 1 to 30). The query performance is shown in Figure 6.10; the modified SM algorithm has again good speed-up performance.

**Multi-Node Speed-Up.** The local speed-up tests show that we can benefit from adding partitions and threads up to the number of cores (4) on the system. The next set of speed-up experiments focuses on adding nodes to the cluster ("scaling out") while the number of partitions per node has been fixed to four.

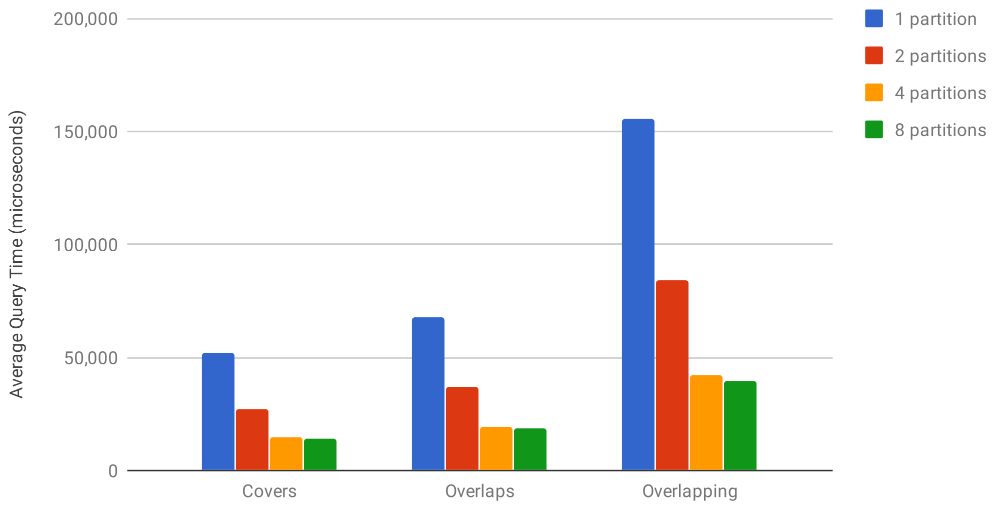


Figure 6.10: Speed-up performance of the modified SM algorithm for a TPC-H generated dataset.

Figure 6.11 shows the first cluster speed-up experiment. The synthetic datasets $R$ and $S$ have 100,000 records, and time range of 10,000 instances, with $\lambda_R = 10$, $\lambda_S = 10$) and duration $d_R = 20$ and $d_S = 10$. The query execution times improve as the number of cluster nodes is increased from 1 to 8. However, the performance gain is slightly less than when adding threads in a single node because the network traffic from global partitioning starts to impact performance while increasing the cluster's aggregate processing power. Figures 6.12, 6.13, and 6.14 show cluster speed-up results while changing cardinality, duration and record size respectively. Again, the modified SM algorithm shows good speed-up for all three queries in the cluster environment.

### 6.3.2 Scale-Up Non-Spilling Experiments

Scale-up experiments demonstrate how the system handles large data by keeping the data size the same per partition as the number of partitions is increased. We first run

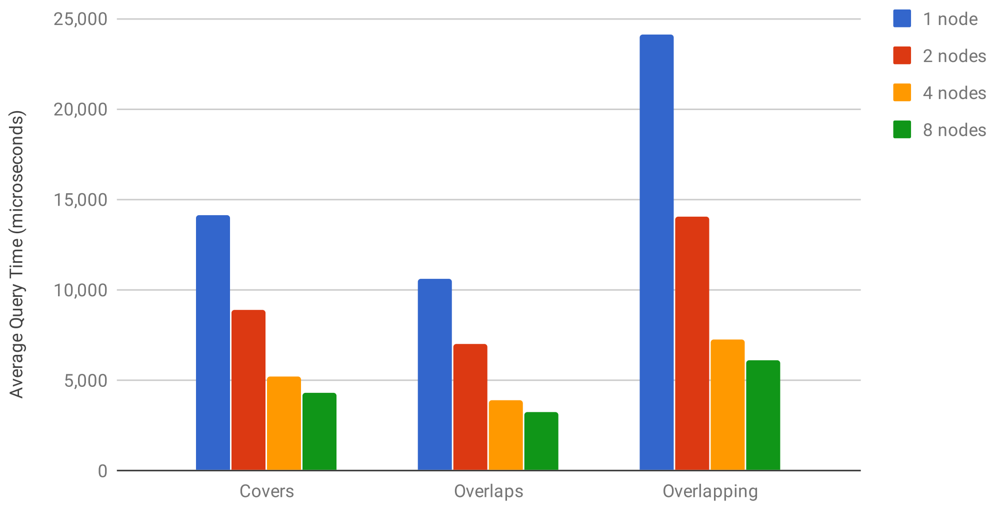


Figure 6.11: Speed-up performance of the modified SM algorithm on a cluster with synthetic data ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 20$ and $d_S = 10$).

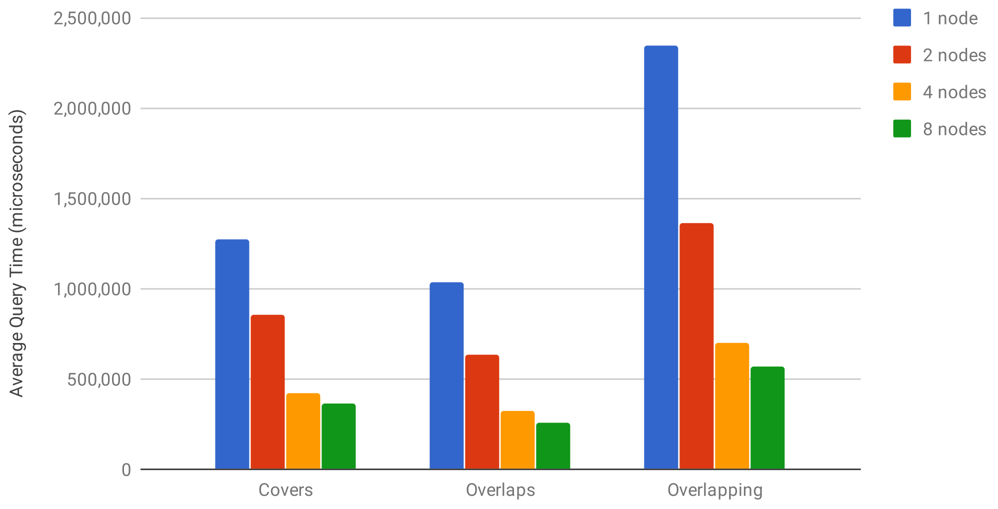


Figure 6.12: Speed-up performance of the modified SM algorithm on a cluster with synthetic data ($\lambda_R = 100$, $\lambda_S = 100$, $d_R = 20$ and $d_S = 10$).

scale-up experiments on a single node. In the non-spilling scale-up case, we are interested in how the CPU is scaling by adding threads as more fixed-size partitions are assigned. We also run scale-up experiments for multiple nodes with a fixed number of partitions per

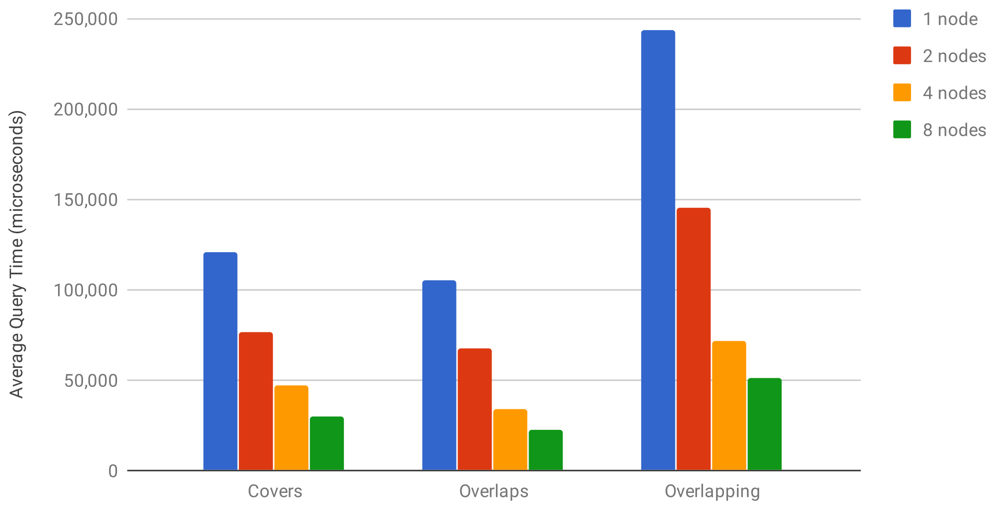


Figure 6.13: Speed-up performance of the modified SM algorithm on a cluster with synthetic data ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 200$ and $d_S = 100$).

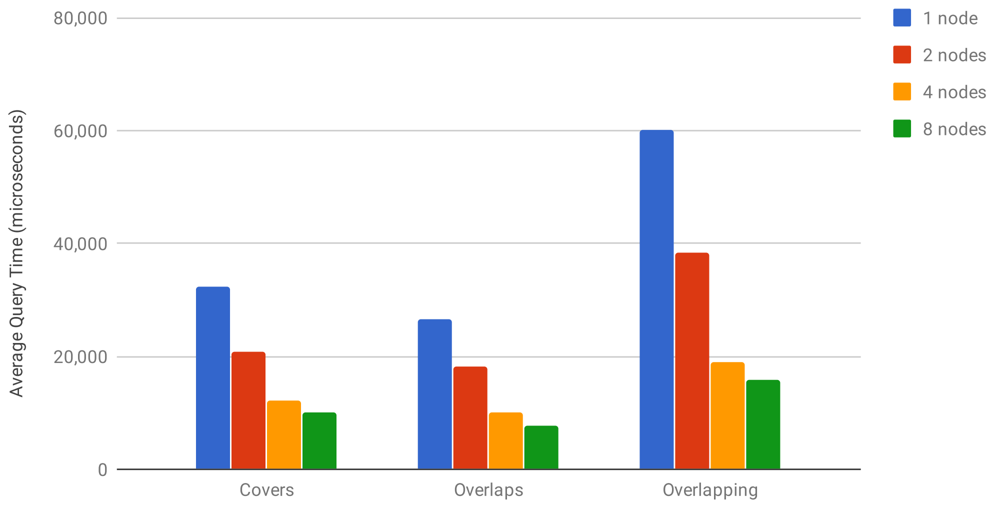


Figure 6.14: Speed-up performance of the modified SM algorithm on a cluster with synthetic data using large records (2296 bytes) ($\lambda_R = 10$, $\lambda_S = 10$, $d_R = 20$ and $d_S = 10$).

node. A single partition in the $R$ and $S$ datasets has 10,000 records, a time range of 1,000, a density of $\lambda_R = 10$ and $\lambda_S = 10$, and duration of $d_R = 20$ and $d_S = 10$. Figure 6.15 shows how the modified SM algorithm scales up on a single node (with 4 cores) using one to

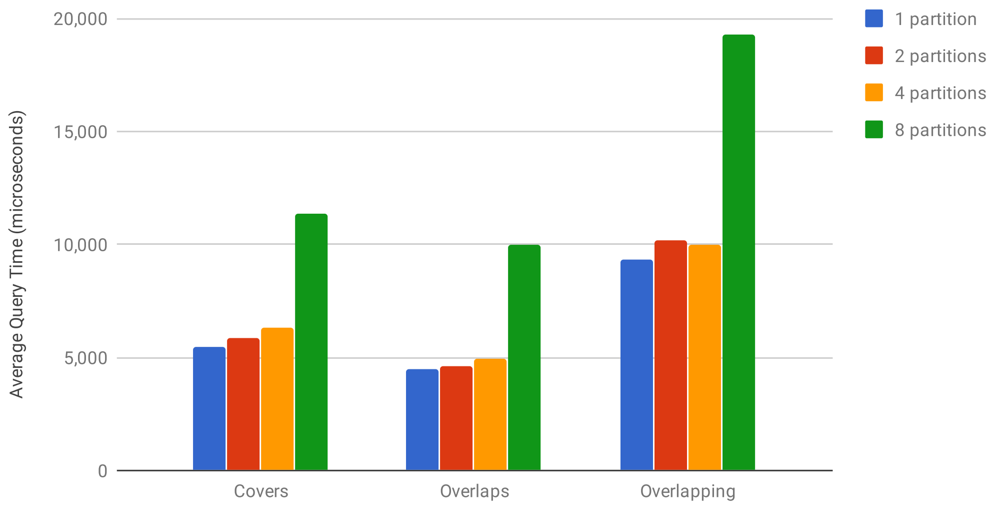


Figure 6.15: Single node scale-up while increasing the number of equal sized partitions (1, 2, 4 and 8).

eight partitions. As it can be seen in the Figure, the modified SM algorithm exhibits good scaling performance up to four partitions, since the node has four dedicated cores and uses hyperthreading for the case of eight partitions. As the query workload doubles from four to eight partitions while the number of cores stays constant at four, the query time doubles from four to eight partitions.

Figure 6.16 shows the multi-node scale-up experiment using one node (four partitions), two nodes (eight partitions), four nodes (16 partitions), and eight nodes (32 partitions). The queries on two or more nodes include network data transfer between nodes to arrange the data for local join processing. This additional network traffic increases the query time. However, overall the modified SM algorithm exhibits good scale-up performance in a cluster, non-spilling environment.

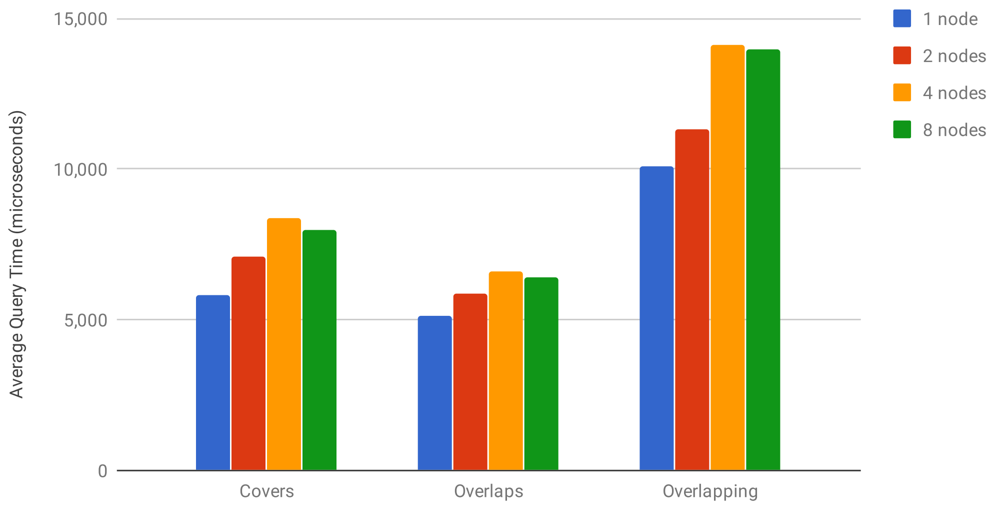


Figure 6.16: Cluster scale-up for 1, 2, 4, and 8 nodes; each node holds four equal sized partitions.

### 6.3.3 Scale-Up Spilling Experiments

The scale-up spilling experiments examine how the modified SM algorithm performs with limited memory. Since we are interested in the I/O performance, the available memory for the join processing is limited to 2MB. Further, the cluster configuration has been updated to map one partition to a single disk. Each partition of dataset $R$ and $S$ has 100,000 records, a time range of 10,000, a density of $\lambda_R = 10$ and $\lambda_S = 10$, a duration of $d_R = 200$ and $d_S = 100$, and a record size of 2,222 bytes. As partitions are added, the cumulative time range is increased accordingly.

For the local scale-up experiments, we only show results for one and two partitions since a node is limited to only two physical disks. The results of the single node scale-up experiments are shown in Figure 6.17. The modified SM algorithm shows good local scale-up spilling behavior for all three interval join conditions.

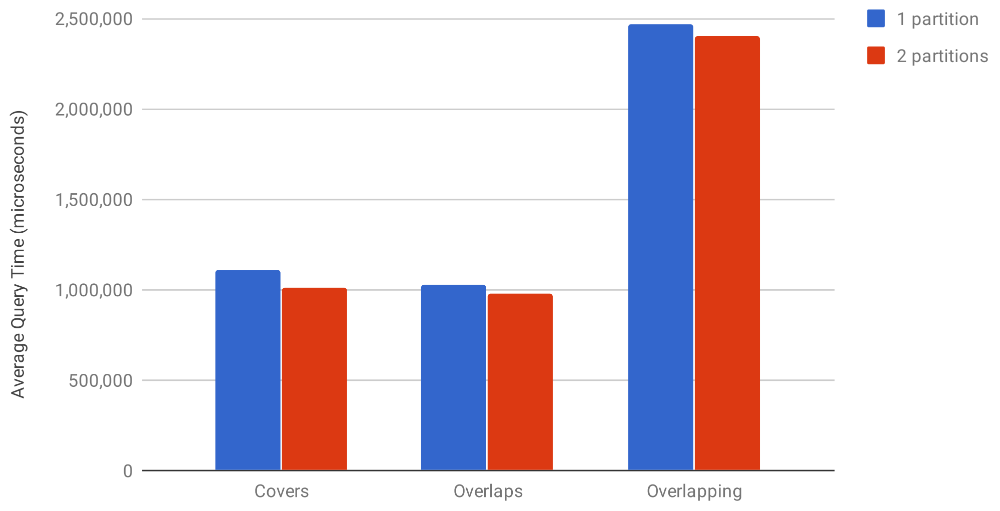


Figure 6.17: Scale-up from 1 to 2 partitions on a single node with large records and spilling.

Figure 6.18 shows the multi-node scale-up experiment which allocated one partition per node, as follows: one node (one partition), two nodes (two partitions), four nodes (four partitions), and eight nodes (eight partitions). Again, the modified SM algorithm shows good scale-up from one node to eight nodes.

## 6.4 Conclusions

In this chapter, we showed how to implement four Allen's relations, namely: *covers*, *covered-by*, *overlaps* and *overlapped-by* by modifying the sort-merge interval join algorithm. The algorithm modifications focus on the edge cases that arise from using Allen's interval algebra for the join condition. The experimental evaluation demonstrated that the modified SM algorithm exhibits good speed-up and scale-up performance for the above four Allen's interval joins. We further summarized changes that can be applied to the other interval join algorithms (FS, TS, OIP and DIP) so as to support the four Allen's relations.

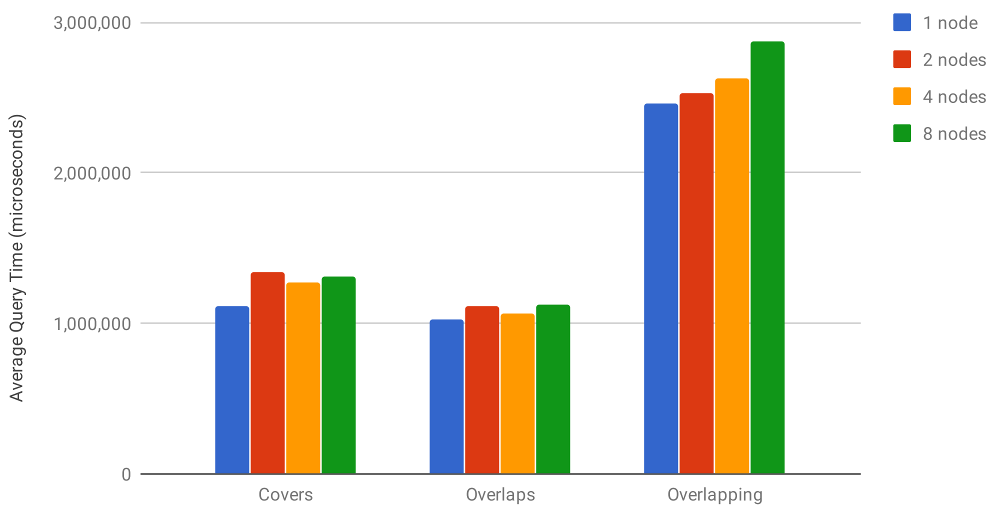


Figure 6.18: Scale-up from 1, 2, 4, and 8 nodes where each nodes holds on partition with large records and spilling.

# Chapter 7

# A Scalable Parallel XQuery Processor

## 7.1 Introduction

There are various native open-source XQuery processors (Saxon [35], Galax [28], etc.), they have been optimized for single-node processing and do not support scaling to many nodes. To create a scalable XQuery processor, one could 1) add scalability to an existing XQuery processor, 2) start from scratch, or 3) extend an existing scalable query framework to support XQuery. Unfortunately, existing XQuery processors would require extensive rewriting of their core architecture features to add parallelism. Similarly, building an XQuery processor from scratch would involve the same complex scalable programming (some unrelated to the XML data model). The last option, extending an existing scalable framework to support XQuery, seems advantageous since it combines the benefits of proven parallel technology with a shorter time to implementation.

Among the several scalable frameworks available, one could use a relational parallel database engine and take advantage of its mature optimization techniques. However, using a framework entails the overhead of translating the data/queries to the relational model and back to XML; moreover, long XML path queries may result in many joins. Another approach is to build an XQuery processor on top of the MapReduce [22] framework. Examples include ChuQL [37], which is a MapReduce extension to XQuery built on top of Hadoop [15], and HadoopXML [21], which combines many XPath queries into a few Hadoop MapReduce jobs. Similarly, Apache MRQL [27] translates XPath queries into an SQL-like language implemented through MapReduce operators. However, these Hadoop-based approaches are limited in that they can only use the few MapReduce operators available (i.e., map, reduce, and combine).

Recently, frameworks have been proposed that generalize the MapReduce execution model by supporting a larger set of operators to create parallel jobs (including Hyracks [12], Spark [49], and Stratosphere [1]). Such ”dataflow” systems [9] typically include flexible data models supporting a wide range of data formats (relational, semi-structured, text, JSON, XML, etc.) which makes them easy to extend. In this chapter, we utilize Hyracks as our parallel framework and use Algebricks [13], a language agnostic compiler toolbox, to implement XQuery.

Our implementation is available as open source at the ASF [8]. We have performed an experimental evaluation using a large (500GB) real dataset (an NOAA weather dataset from [41]) and various selection, aggregation, and join XML queries that show the efficiency of our XQuery processor, both in terms of speed up and scale up.

The rest of this chapter is organized as follows: Section 7.2 reviews current approaches for querying large XML data repositories while Section 7.3 covers the Apache VXQuery software stack that builds upon the Hyracks and Algebricks framework and how the data model, parser, and runtime were extended for XQuery support. Given the specifics of XQuery, we had to extend existing Algebricks rewrite rules and introduce new ones; this discussion appears in Section 7.4. Finally, Section 7.5 presents the results of our experiments on Apache VXQuery's performance as well as a comparison with two open-source XML processors: the single-threaded SaxonHE and the parallel Apache MRQL.

## 7.2 Related Work

Hadoop [6] provides a framework for distributed processing based on the MapReduce model. The MapReduce model leaves a significant implementation burden on the application programmer. As a result, a number of languages have been proposed on top of Hadoop (e.g., Hive [46], PigLatin [42], and Jaql [10]); however, popular high-level MapReduce languages do not support the XML data model. Recent approaches to close this gap include ChuQL, Apache MRQL, HadoopXML, and Oracle XQuery for Hadoop.

ChuQL [37] extends XQuery to include MapReduce support for processing native XML on Hadoop. In ChuQL, a MapReduce expression is included as an XQuery function, allowing the query writer to specify the MapReduce job definition in XQuery. In contrast, VXQuery hides all parallel processing details from the query writer while still using standard XQuery constructs.

Apache MRQL (MapReduce Query Language) [26, 27] is an SQL-like language designed to run big data analysis tasks. The language supports parsing XML data from Hadoop through a *source* expression defining the XML parser, XML file, and XML tags. The XML parser processes the XML file and returns all elements matching these tags; Apache MRQL then translates these elements into the Apache MRQL data model. Each query is translated to an algebra expression for the Apache MRQL cost-based optimizer, which builds upon known relational query and MapReduce optimization techniques. The algebra uses a small number of physical operators to create a more efficient MapReduce job than directly writing it using the MapReduce operators.

HadoopXML [21] processes a single large XML file with a predetermined set of queries (each currently in a subset of XPath). The engine identifies the query commonalities (paths that are common) and executes those once; it then shares the common results and augments them with the non-common parts per query. This processing is performed using MapReduce jobs. When a query is executed, the query optimizer determines the optimum number of jobs to execute the requested query.

Recently, Oracle released Oracle XQuery for Hadoop (OXH) [47], which runs XQuery data transformations by translating them into a series of MapReduce jobs.

In summary, the above approaches share the MapReduce framework and are thus limited to using only the available MapReduce operators. Apache VXQuery differs in that it is built on top of a more general scalable framework (Hyracks) and can match XQuery computational tasks to Hyracks' richer existing operators (e.g., join); this matching, in turn, provides better performance. As will be seen in our experimental section, our rewrite rules, together with Hyracks' efficiency, provides over twice the performance of approaches that perform XML processing on top of MapReduce.

PAXQuery [18] implements XQuery top of Stratosphere [1] (a dataflow system that is similar to Hyracks). The system translates XQuery queries into an internal XQuery algebra and then into Parallelization Contracts (PACTs) while Apache VXQuery translates the query into a language agnostic algebra (Algebricks) and then into a Hyracks job for execution. PAXQuery builds on previous unnesting optimizations for tuple-based XQuery algebras [11, 23, 40, 45]. Since Apache VXQuery also uses a tuple-based algebra, the same optimization techniques can be applied to the Algebricks query plans. PAXQuery was not available for comparison as of the writing of this chapter. Similarly, the Apache MRQL group is currently working on supporting Apache MRQL on top of Apache Flink [5] (which evolved from the Stratosphere project), but at the time of this writing that implementation was still under development.

## 7.3 Apache VXQuery's Stack

Apache VXQuery's software stack can be represented in three layers, as shown in Figure 7.1. The top layer, Apache VXQuery, forms an Algebricks logical plan based on parsing a supplied XQuery. The initial Algebricks logical plan is then optimized and translated into an Algebricks physical plan that maps directly to a Hyracks job. The

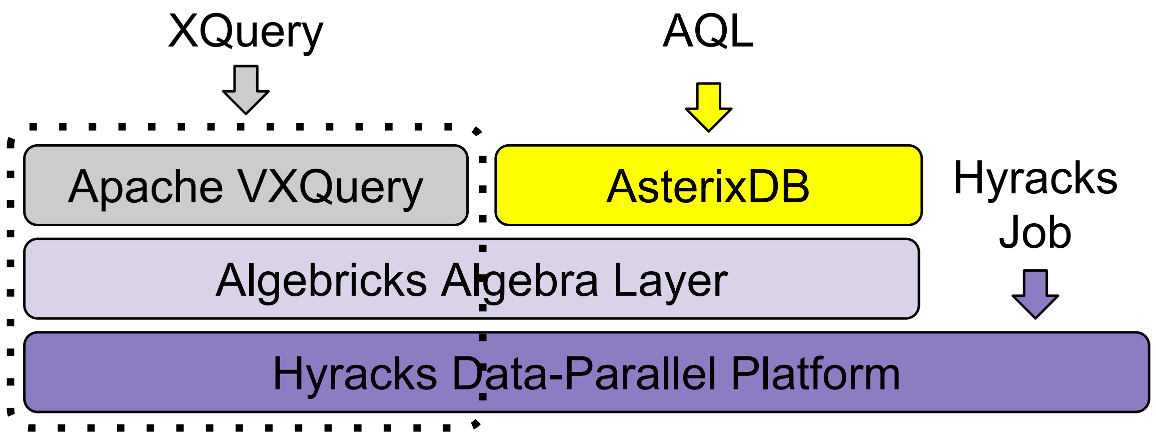


Figure 7.1: The layers of the Apache VXQuery stack.

Hyracks platform executes the job and returns the results. Figure 7.1 also shows AsterixDB [3], another system that uses the Hyracks and Algebricks infrastructure.

Apache VXQuery utilizes additional Algebricks logical operators covered in this chapter. The field names in the query plans are represented by $$ followed by a number in remaining text. The following Algebricks logical operators are commonly used in VXQuery:

The ASSIGN operator executes a scalar expression on a tuple and adds the result as a new field in the tuple.

The AGGREGATE operator executes an aggregate expression to create a result tuple from a stream of input tuples. The result is held until all tuples are processed and then returned in a single tuple.

The UNNEST operator executes an unnesting expression for each tuple, creating a stream of single item tuples.

The SUBPLAN operator executes a nested plan for each tuple input.

The NESTED-TUPLE-SOURCE operator is used as the initial operator in nested plans. The operator links the nested plans with the input to the operator (such as a SUBPLAN) defining the nested plan.

The Algebricks operators are each parameterized with custom expressions. The expressions map directly to specific language functions or support runtime features. Each expression is an instance of one the following expression types: scalar, aggregate, and unnesting. Most operators use a scalar expression while the AGGREGATE and UNNEST operators have their own expression types. The three expression types differ in their input and output cardinalities. Scalar expressions operate on a single value and return a single value. Aggregate expressions consume many values to create a single result. Unnesting expressions consume a single (usually structured) value to create many new values. Correspondingly, the AGGREGATE and UNNEST operators change the cardinality of the tuple stream.

Apache VXQuery extends the language agnostic layer provided by Algebricks to create a scalable XQuery processor. Apache VXQuery provides a binary representation of the XQuery Data Model (XDM) (an example can be found at the Apache VXQuery website [8]), an XQuery parser, an XQuery optimizer, and the data model dependent expressions. VXQuery can process data that is supplied in non-fragmented XML documents partitioned evenly throughout a cluster. A SAX-based XML parser translates the XML documents at runtime into XDM instances. Hyracks base types were extended to build untyped XDM instances for the XQuery node types and the XQuery atomic types. (All XQuery types used are listed in Table 7.1.)

| | |
|---|---|
| Hyracks Base | boolean, byte, short, integer, long, double, float, UTF8 string |
| XQuery Atomic | binary, decimal, date, datetime, time, duration, QName |
| XQuery Node | attribute, comment, document, element, processing instruction, text |

Table 7.1: Apache VXQuery builds on the Hyracks Base types to create the XQuery Atomic and Node data types.

Query evaluation proceeds through the usual steps. The query is parsed into an abstract syntax tree (AST) and is then analyzed, normalized, and translated into a logical plan. The logical plan consists of Algebricks data model independent operators parameterized with Apache VXQuery data model dependent expressions. The logical plan is then optimized using both generic rewrite rules provided by Algebricks and XQuery specific rewrite rules provided by Apache VXQuery (discussed in Section 7.4). After rewriting the logical plan, it is translated into a physical plan and optimized further (physical optimization includes such things as the selection of join methods or the distribution of the plan). Finally the physical plan is translated into a Hyracks job that is executed. Similar to Algebricks operators that have physical representations based on Hyracks operators, Apache VXQuery provides executable functions that implement Apache VXQuery's data model dependent expressions.

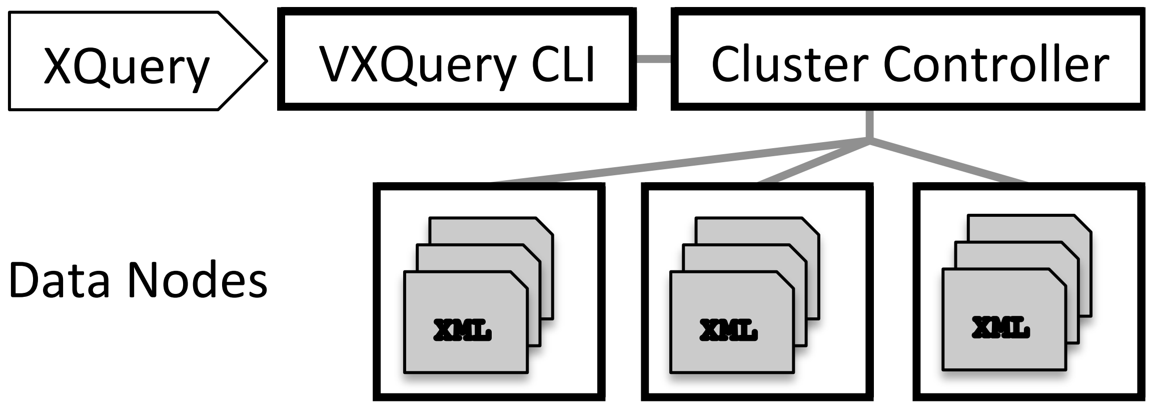


Figure 7.2: The VXQuery cluster configuration.

Special attention is required regarding how the XDM defines a set of items as a sequence. In Apache VXQuery, an XDM sequence can have two forms: a *sequence item* or a *tuple stream*. A sequence item holds all the values in a single tuple field; a tuple stream represents the sequence using a field with the same name in multiple tuples. To switch between these representations, we provide the *iterate* and the *create_sequence* expressions. The *iterate* unnesting expression works with Algebricks' UNNEST operator to convert a tuple field that holds a sequence item into a stream of individual tuples. The *create_sequence* aggregate expression executes within Algebricks' AGGREGATE operator to consume a tuple stream and create a sequence item for inclusion in a single output tuple. The two expressions are used during the logical rewrite process to switch between formats to enable further optimization rules to be applied to the query plan.

At runtime, the Apache VXQuery cluster processes a query using the Apache VXQuery Client Library Interface (CLI), a Hyracks Cluster Controller, and some Hyracks Data Nodes (as shown in Figure 7.2). The process starts with a user submitting an XQuery statement to the Apache VXQuery CLI for parallel execution. The CLI parses and optimizes the query and submits the generated Hyracks job to the cluster controller, which manages and distributes tasks to each of the data nodes for evaluation. Each data node contains XML files, an XML parser, and the XQuery runtime expressions used to evaluate the node's tasks. Finally, the cluster controller collects the data nodes' results and sends the result back to the Apache VXQuery CLI, which returns the result to the user.

## 7.4 Rewrite Rules

Algebricks provides generic rules for both Logical-to-Logical and Logical-to-Physical plan optimizations. These rules include actions that consolidate, push down, and/or remove operators based on the operators' properties and the query plan. In addition, to build the XQuery optimizer we needed to implement XQuery-specific rules; these rules fall into two categories. The *Path Expression Rewrite Rules* attempt to remove subplans that are introduced by the unnesting required to evaluate path expressions. The *Parallel Rewrite Rules* transform the plan to enable parallel evaluation for specific XQuery constructs (aggregation, join, and data access) using both pipelined and partitioned parallelism.

### 7.4.1 Path Expression Rewrite Rules

The normalization phase of query translation introduces explicit operations into the query plan that ensure the correctness of the plan (for example, sorting to maintain document order). However, some of these operations may not be required based on knowledge of the structure of the plan and the implementation of operators and expressions. The following XML segment is based on the sample XML tree from the W3Schools tutorial [48] for XQuery and will be used to outline the path expression rewrite rules.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
  <book id="1" category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
  </book>
  <book id="2" category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
  </book>
   ...
</bookstore>
```

Consider the following simple query.

```
doc("book.xml")/bookstore/book
```

The query reads data from the document *book.xml* located in the file system using the XQuery *doc* function. Next, the first child path step expression ("/bookstore") is applied to the document node. Three stages are used when applying the child path step expression to a tuple: each input node is iterated over, any matching child nodes are put into a single

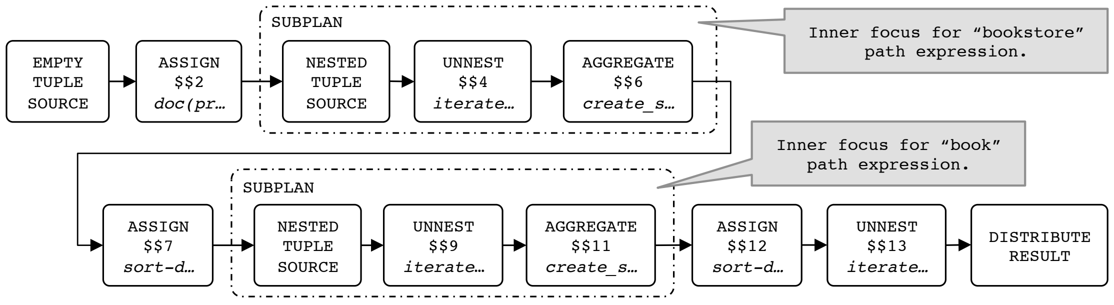


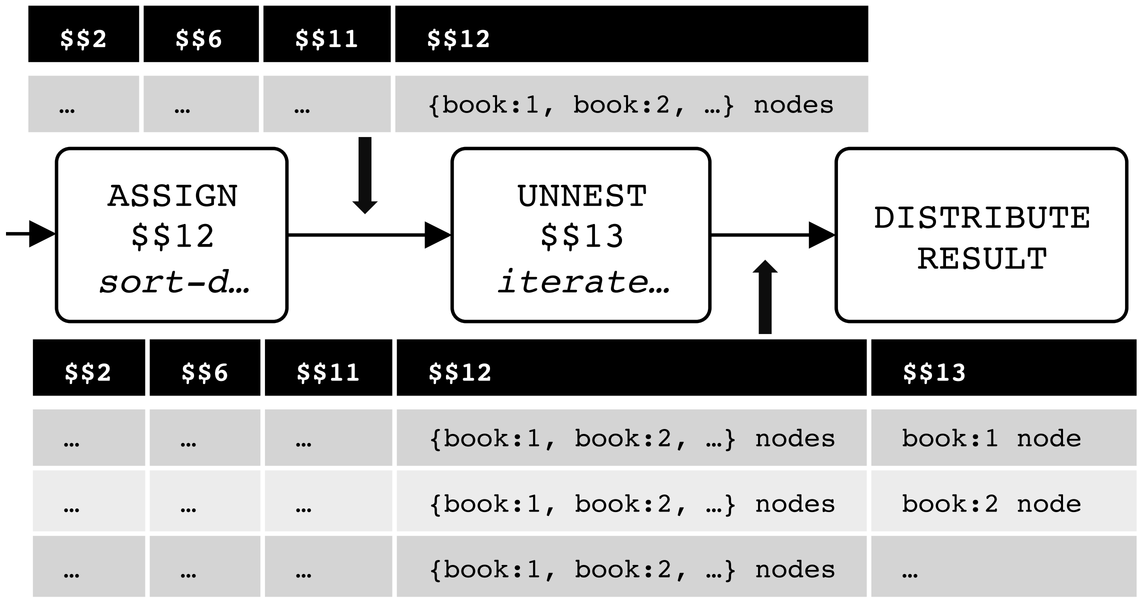
Figure 7.3: Example query dataflow DAG before applying rewrite rules.

sequence, and the sequence is then sorted in document order. The same path step process is applied to each resulting "bookstore" element node for the second child path step expression ("/book"). Finally, each "book" element node is then returned in the final query result.

VXQuery creates the initial plan shown below (after removing unused variables); here the curly braces represent nested plans that are executed for each of the SUBPLAN's input tuples. Schematically this plan, which is read bottom-up, corresponds to the dataflow DAG in Figure 7.3. The DAG is a single path of execution in this case. Each DAG is initialized with an EMPTY-TUPLE-SOURCE operator and collects its results into a DISTRIBUTE-RESULT operator.

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:iterate($$12) )
ASSIGN( $$12:sort-distinct-nodes-asc-or-atomics($$11) )
SUBPLAN {
  AGGREGATE( $$11:create_sequence(child(treat($$9, element_node),
    "book")) )
  UNNEST( $$9:iterate($$7) )
  NESTED-TUPLE-SOURCE
}
ASSIGN( $$7:sort-distinct-nodes-asc-or-atomics($$6) )
SUBPLAN {
  AGGREGATE( $$6:create_sequence(child(treat($$4, element_node),
    "bookstore")) )
  UNNEST( $$4:iterate($$2) )
  NESTED-TUPLE-SOURCE
}
ASSIGN( $$2:doc(promote(data("books.xml"), string) )
EMPTY-TUPLE-SOURCE
```

The plan's EMPTY-TUPLE-SOURCE operator creates the initial empty tuple. The *doc* expression in the ASSIGN operator (line 15) returns a document node using the string URI argument and adds a new field – $$2:document node – to the tuple. The *promote* and *data* expressions ensure the *doc* URI argument will be a string. The SUBPLAN operator (line 10) uses a nested plan to implement the first and second stages of the /bookstore path step. The subplan's nested plan ensures the correct dynamic context for the path step and provides an "inner focus" to evaluate the expression on each item in the sequence for the next step (if any). The NESTED-TUPLE-SOURCE operator (line 13) connects the nested plan to the SUBPLAN's input dataflow. The input tuple is passed on to the UNNEST operator (line 12) where each $$2:document item is iterated over and added as $$4:document. For the



| $$2 | $$6 | $$11 | $$12 |
|---|---|---|---|
| … | … | … | {book:1, book:2, …} nodes |

| $$2 | $$6 | $$11 | $$12 | $$13 |
|---|---|---|---|---|
| … | … | … | {book:1, book:2, …} nodes | book:1 node |
| … | … | … | {book:1, book:2, …} nodes | book:2 node |
| … | … | … | {book:1, book:2, …} nodes | … |

Figure 7.4: Dataflow segment for the last UNNEST operator.

root path step expression, there is only one item in the sequence. The inner focus is closed with AGGREGATE (line 11) processing all SUBPLAN tuples using the create_sequence (child(treat($$4, element_node), ”bookstore”)) expression. The expression ensures that *child* expression’s argument is of type ”element_node” (*treat*), finds all ”bookstore” child nodes (*child*), and creates a sequence of all the results (*create_sequence*). The resulting tuple now holds two fields: $$2:document node and $$6:”bookstore” node. All the SUBPLAN variables are discarded except the final operator’s result (in this case, AGGREGATE $$6). The third stage of the path step is completed though the ASSIGN operator (line 9) with sort−distinct−nodes−asc−or−atomics($$6). The expression creates a new field with nodes that are in document order and duplicate free from $$6:”bookstore” node. Since there is only one item, the ”bookstore” node is copied over to $$7.

The next SUBPLAN (line 4) creates the inner focus for the /book path step expression. Similar to the /bookstore path step, the nested plan iterates over the input tuples and saves all child nodes {book:1, book:2, ...} to $$11. The ASSIGN (line 3) ensures document order in the child "book" node sequence by removing duplicates and sorting the sequence. Finally, each item in $$12:{book:1, book:2, ...} is unnested by UNNEST (line 2) to create a tuple stream for the DISTRIBUTE-RESULT operator (line 1). See Figure 7.4 for a graphic representation of the tuples before and after this UNNEST operator.

The initial query plan is inefficient and can be improved in several ways: we can (i) remove the computationally expensive sort operators (as document order is not changed by any of the other operators) and (ii) remove the SUBPLAN operators (since each SUBPLAN corresponds to a simple step expression the inner focus is not required). After these optimization rules, the plan can be cleaned further by (iii) enabling unnesting (improves operator efficiency) and (iv) merging the path step unnesting operators (reduces number of operators). Due to space constraints, a detailed explanation of these path step expression rewrite rules can be found in our technical report [19].

After applying these rewrite rules recursively to the sample query plan, the resulting plan only uses only a single UNNEST operator to represent the two child path step expressions. The path expression rewrite rules create the following updated sample query plan:

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:child(child($$2, "bookstore"), "book") )
ASSIGN( $$2:doc("books.xml") )
```

EMPTY–TUPLE–SOURCE

### 7.4.2 Parallel Rewrite Rules

After applying the path expression rewrite rules, the plan is optimized for parallel XQuery processing. Hyracks allows for both pipelined and partitioned parallelism. We thus introduce rules to enable the use of Hyracks' parallel execution features. To take advantage of pipelining in Apache VXQuery, we create fine grained data items. For example, the DATASCAN operator introduced next does not compute a whole collection at once, but instead it computes chunks that can be fed to the remaining operators. As a side effect, the needed buffer size (Hyracks' frame) is reduced between the operators in the pipeline. To introduce partitioned parallelism, we use partitioned data access for physically partitioned data ,and we use partitioned parallel algorithms for join and aggregation.

#### Introduce the DATASCAN Operator

To query a collection of XML documents, XQuery defines a function called *collection* that maps a string to a sequence of nodes. Apache VXQuery interprets the string as a directory location, reads in data from the files in the directory, and returns all nodes as a single sequence value. Since the collection query considers many documents, it can produce a large number of query results. Instead of gathering all nodes into a single sequence, we would like to send one node at a time through the pipeline. To avoid this problem, we combine the *collection* expression with an *iterate* expression (typically inserted because of a path step or a for clause) to split the large document sequence into many single document tuples, thus reducing the size of the materialized result. Below is a sample *collection* query similar to the previous single document query example, followed by the query plan generated after the path expression rules have been applied.

```
collection("/books")/bookstore/book
```

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:child( child($$4, "bookstore"), "book") )
UNNEST( $$4:iterate($$2) )
ASSIGN( $$2:collection(promote(data("/books"), string)) )
EMPTY-TUPLE-SOURCE
```

The path expression rules have conveniently moved an UNNEST *iterate* above the ASSIGN *collection*, creating a stream of XML document tuples. Algebricks offers a DATASCAN operator to directly create a stream of tuples based on a data source. Since *collection* already defines the data source, the DATASCAN operator can be used to replace UNNEST *iterate* and ASSIGN *collection*. The updated query plan is:

```
DISTRIBUTE-RESULT( $$13 )
UNNEST( $$13:child( child($$4, "bookstore"), "book") )
DATASCAN( collection("/books"), $$4 )
EMPTY-TUPLE-SOURCE
```

The finer grained tuples reduce the buffer size between operators during the query execution. Note that the above rewrite rule allows Apache VXQuery to process any amount of XML data provided that the largest XML document in the collection can fit in Hyracks' frame size. This constraint can be further reduced to the largest subtree under the query path expression. This rule is possible when the UNNEST *child* expression is the consumer of a DATASCAN operator. The *child* expression can be merged into the DATASCAN operator to provide even smaller tuples. The query plan is updated to show that the DATASCAN operator has a third argument specifying the child path expression; the updated DATASCAN operator includes the path expression within the collection:

```
DISTRIBUTE−RESULT( $$4 )
DATASCAN( collection("/books"), $$4, "/bookstore/book" )
EMPTY−TUPLE−SOURCE
```

In addition to the improved pipeline, the DATASCAN operator offers a way to introduce partitioned parallelism simply by specifying Apache VXQuery's partition details to this operator. In Apache VXQuery, data is partitioned among the cluster nodes. Each node has a unique set of XML documents stored under the same directory specified in the *collection* expression. The Algebricks' physical plan optimizer uses the details of these partitioned data properties to distribute the query execution. For example, path "/books" defined in the *collection* expression is located on each node and represents a unique set of XML documents for the query. These partition properties are added to the DATASCAN operator although these properties is not shown in the query plan. Adding these properties allows Apache VXQuery to achieve partitioned parallel execution without any parallel programming.

## Replace Scalar with Aggregate Expressions

The XQuery aggregate expressions (*avg*, *count*, *max*, *min*, and *sum*) use a default scalar implementation in a normalized query plan. This plan implies that the whole result is first stored in a sequence which is then processed to produce the aggregate. Instead of materializing the sequence, we can match the XQuery aggregate expression with an Algebricks AGGREGATE operator. When the Algebricks AGGREGATE operator is used with an XQuery aggregate expression, the result will be incremental aggregation instead of materializing all records in the operator's buffer. Consider a query that counts the number of book elements in an XML collection and the query plan produced using the previous rules:

```
count(
  for $x in collection("/books")/bookstore/book
  return $x
)
```

```
DISTRIBUTE-RESULT( $$17 )
UNNEST( $$17:iterate($$16) )
ASSIGN( $$16:count($$15) )
SUBPLAN {
  AGGREGATE( $$15:create_sequence($$4) )
  DATASCAN( collection("/books"), $$4, "/bookstore/book" )
  NESTED-TUPLE-SOURCE
}
```

```
EMPTY-TUPLE-SOURCE
```

The XQuery aggregate expression *count* is within an ASSIGN operator (line 3). The SUBPLAN finds the bookstore nodes and uses an AGGREGATE operator (line 5) to store them in a sequence. However, there is no UNNEST directly above the SUBPLAN (as shown in our technical report for the path expression rewrite rules) and thus the SUBPLAN cannot be removed. However, the scalar *count* expression applies its calculation on the produced XQuery sequence to create $$16's value. Instead, the aggregate *count* expression can replace the *create_sequence* within the Algebricks AGGREGATE operator, thus performing aggregation incrementally instead of first generating a large XQuery sequence. The updated query plan becomes:

```
DISTRIBUTE-RESULT( $$17 )
UNNEST( $$17:iterate($$16) )
SUBPLAN {
  AGGREGATE( $$16:count($$4) )
  DATASCAN( collection("/books"), $$4, "/bookstore/book" )
  NESTED-TUPLE-SOURCE
}
EMPTY-TUPLE-SOURCE
```

The new plan keeps the pipeline granularity and enables partitioned aggregation processing. An additional Apache VXQuery rule annotates the AGGREGATE operator with local and global aggregate expressions, enabling the use of Algebricks' support for two-step aggregation: each partition calculates its local aggregate result on its data and then transmits the result to a central partition for the global computation. As a result, partitioning also reduces communication thus improving parallel processing efficiency.

### Introduce the JOIN Operator

In XQuery, two distinct datasets can be connected (matched) through a nested *for* loop. The normalized query plan follows the same nested loop, which can be very expensive; we can do better by using a relational-style join. We note that Algebricks provides a JOIN operator as well as a set of language independent rewrite rules to optimize generic query plans. We can thus use these provided rewrite rules to translate the nested loop plan in to a join plan. Consider a query that takes two bookstores (Ann and Joe) and finds books with the same title, and its query plan below:

```
for $r in collection("/ann−books")/bookstore/book
for $s in collection("/joe−books")/bookstore/book
where $r/title eq $s/title
return $r
```

```
DISTRIBUTE−RESULT( $$32 )
UNNEST( $$32:iterate($$27) )
SELECT( boolean(value−eq($$27, $$28)) )
ASSIGN( $$28:data(child($$26, "title")) )
ASSIGN( $$27:data(child($$13, "title")) )
DATASCAN(collection("/joe−books"),$$26,"/bookstore/book")
DATASCAN(collection("/ann−books"),$$13,"/bookstore/book")
```

```
EMPTY−TUPLE−SOURCE
```

In this example each dataset is identified and accessed by a DATASCAN operator while the SELECT operator contains the condition for connecting the two datasets (which effectively will become the join condition). The two ASSIGN operators (line 4 and 5) find the title child node and return the atomic value of the node. Three Algebricks language-independent rules are used to introduce the JOIN operator. The first rule converts the nested DATASCAN operator into a cross product; it identifies that each data source is independent and adds the JOIN operator with a condition of true (basically a cross-product). The second Algebricks rule manipulates the DAG to push down operators that only affect one side of the join branch (selection, assign, etc). The third rule then merges the SELECT and JOIN operators so the join condition (from the SELECT) is within the JOIN operator. In the final plan, the JOIN operator has one branch from each data source, which allows each branch to be processed locally and then joined together globally.

```
DISTRIBUTE−RESULT( $$32 )
UNNEST( $$32:iterate($$27) )
JOIN( boolean(value−eq($$27, $$28)) )
{
  ASSIGN( $$28:data(child($$26, "title")) )
  DATASCAN(collection("/joe−books"),$$26,"/bookstore/book")
  EMPTY−TUPLE−SOURCE
} {
  ASSIGN( $$27:data(child($$13, "title")) )
  DATASCAN(collection("/ann-books"),$$13,"/bookstore/book")
  EMPTY-TUPLE-SOURCE
}
```

Going from here to the final logical plan does not require any custom Apache VXQuery rules, but the physical plan needs more information to choose the most efficient join algorithm. The equality comparison in our sample query allows the use of a more efficient partition-based algorithm. If Algebricks understands the condition characteristics, it can choose an optimal hash-based join. For Algebricks to identify the join condition, this condition must be represented by a boolean Algebricks expression, in this case the Algebricks' *equal* expression for a hash-based join. (Other Algebricks generic expressions include *and, or, not, less than, greater than, less than or equal, greater than or equal, not equal.*) As the extraction of the XQuery's Effective Boolean Value of the value-comparison in the previous plan (boolean(value−eq(...))) is equivalent to Algebricks' *equal* expression, we convert one to the other thus enabling Algebricks to identify the join. After running the physical optimization rules, the Algebricks expression is converted back to the original XQuery expressions for runtime evaluation. As a result, Hyracks will now use a Hybrid-Hash Join algorithm to achieve efficient partitioned parallelism.

## 7.5 Apache VXQuery Performance

To examine the scalability of our XQuery implementation we have performed an experimental evaluation using publicly available weather XML data. We have also performed a comparison of Apache VXQuery with two open-source XML processors: Saxon [35] and Apache MRQL [7, 27].

### 7.5.1 Weather Data

The NOAA website [41] offers weather data via an XML-based web service. For our queries, we chose the Global Historical Climatology Network (GHCN)-Daily dataset that includes daily summaries of climate recordings. The core data fields report high and low temperatures, snowfall, snow depth, and rainfall. The complete data definition and field list can be found on NOAA's site [41]. The date, data type, station id, value, and various attributes (i.e., measurement, source, and quality flags) are included for each weather report. In addition, a separate web service provides additional station data: name, latitude, longitude, and date of first and last reading. The datasets used had four different sizes, ranging from 500MB up to 500GB.

### 7.5.2 Queries

Here we consider three basic types of XQuery queries: selection, aggregation and join. The complete benchmark results include additional query variations, but due to space constraints some are shown only in our technical report [19]. For consistency, the queries below follow the same numbering as [19].

*Selection:* Query 7.2 finds all readings that report an extreme wind warning. Such warnings occur when the wind speed exceeds 110 mph. (The wind measurement unit, tenths of a meter per second, has been converted to miles per hour.)

```
for $r in collection("/sensors")/dataCollection/data
where $r/dataType eq "AWND"
  and decimal(data($r/value)) gt 491.744
return $r
```

Query 7.2: Extreme Wind Warming

*Aggregation:* Query 7.4 finds the highest recorded temperature in the weather data set. The Celsius temperature is reported in tenths of a degree.

```
max(
  for $r in collection("/sensors")/dataCollection/data
  where $r/dataType eq "TMAX"
  return $r/value
) div 10
```

Query 7.4: Highest Recorded Temperature

*Join:* Query 7.6 finds the highest recorded temperature (TMAX) for each station for each day during the year 2000.

```
for $s in collection("/stations")/stationCollection/station
for $r in collection("/sensors")/dataCollection/data
where $s/id eq $r/station
  and $r/dataType eq "TMAX"
  and year-from-dateTime(dateTime(data($r/date))) eq 2000
return ($s/displayName, $r/date, $r/value)
```

Query 7.6: High Temperature per Station

*Join and Aggregation:* In Query 7.8 we join two large collections, one that maintains the daily minimum temperature per station and one that contains the daily maximum temperature per station. The join is on the station id and date and finds the daily temperature difference per station and returns the average difference over all stations.

### 7.5.3 Experimental Results

Our performance study explores Apache VXQuery's ability to scale locally with the number of cores and then in a cluster with the number of nodes. In the single node

```
avg(
  for $r_min in collection("/sensors_min")/dataCollection/data
  for $r_max in collection("/sensors_max")/dataCollection/data
  where $r_min/station eq $r_max/station
    and $r_min/date eq $r_max/date
    and $r_min/dataType eq "TMIN"
    and $r_max/dataType eq "TMAX"
  return $r_max/value - $r_min/value
) div 10
```

Query 7.8: Average Daily Temperature Differential

tests, the number of data partitions is varied to demonstrate nodes scaling up to the number of available cores. For these tests, partitions represent data splits and each partition has a separate query execution thread. In the cluster tests, the number of partitions per node has been fixed (to one partition per core), and only the number of nodes is varied. The tests were all executed on an eight-node gigabit-connected cluster. Each node has two dual-core AMD Opteron(tm) processors, 8GB of memory, and two 1TB hard drives.

**Single Node Experiments**

Our single node experiments used one cluster node and repeated each query five times. The reported query time is an average of the last three runs. (In our setting, the first two executions are used to warm up the system.) The first single node experiment compares Apache VXQuery with Saxon [36], which is a highly efficient open-source XQuery processor. The freely available Saxon Home Edition (SaxonHE 9.5) is typically limited to a single thread processing data that can fit into one fifth the size of the machine's memory. A group of weather stations were selected to create query results that fit these Saxon data restrictions. The 584MB data set has been partitioned on a single hard drive. The speed-up test keeps the total data set size constant while varying its number of data partitions and corresponding query processing threads.

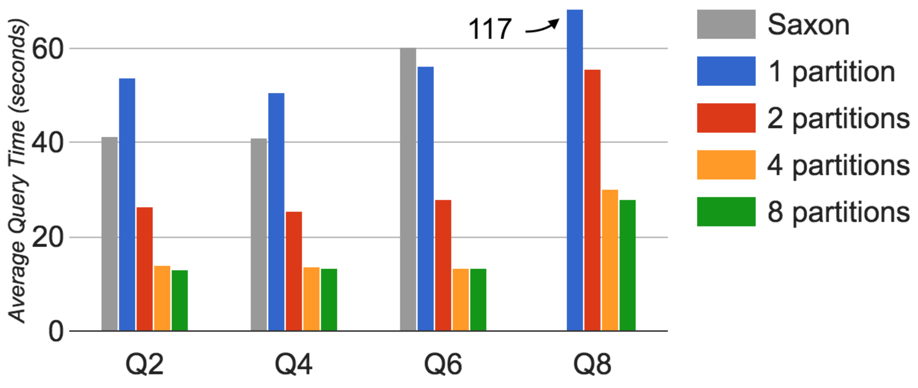


Figure 7.5: Single node speed-up comparison for Saxon and Apache VXQuery (584MB dataset; varying Apache VXQuery partitions).

Figure 7.5 shows the single node speed-up performance results for Apache VXQuery and Saxon. Only a single experiment is shown for Saxon since multi-threading is not available in the SaxonHE 9.5 version. Apache VXQuery outperforms Saxon when it uses two or more partitions. The single partition results are slower due to overhead introduced for parallel and distributed query processing. The join queries (Query 7.6 and 7.8) are translated into hash-based joins for Apache VXQuery, thus giving better performance than Saxon's nested-loop join. Saxon's result for Query 7.8 is not reported since the large number of joined records caused it to never complete (and based on our other results, this query could take several months to complete). For the rest of the queries (Query 7.2, 7.4, and 7.6), when using 4 or more partitions, Apache VXQuery performed on average about 3.5x faster than Saxon. The Apache VXQuery performance for 8 partitions is similar to its 4 partitions performance, which is when the CPU becomes saturated.

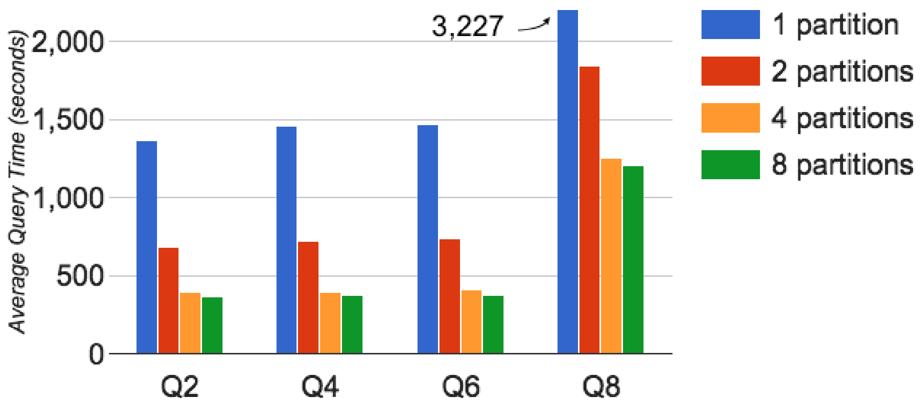


Figure 7.6: Single node Apache VXQuery speed-up (15.2GB dataset).

To further test single node speed-up for Apache VXQuery, we also used a dataset larger than the node's memory (8GB). For this experiment, we used an XML weather data subset, the GCOS Surface Network (GSN) stations containing 15.2GB of XML data. The results appear in Figure 7.6. As with the previous figure, Apache VXQuery scales well up to the node's number of cores (4). Similar to the single node 584MB experiments, the CPU is saturated when using 4 or more partitions. While profiling our experiments, we observed that Apache VXQuery is CPU-bound here, despite the larger data size, due to the overhead of parsing the XML document for each query. The CPU-bound process is also evident from the improvement in performance when increasing threads.

## Cluster Experiments

Based on the single node speed-up results, the cluster experiments used eight nodes and four partitions per node. The first cluster tests used the U.S. Historical Climatology Network (HCN) stations dataset which holds 57GB of XML weather data. This dataset exceeds the available cluster memory when using less than eight nodes. For each experiment, the dataset was equally divided among the nodes participating in the experiment.

The cluster speed-up results for Apache VXQuery (as well as for Apache MRQL, to be discussed later) appear in Figure 7.7; the Apache VXQuery query times are depicted by the circles inside the corresponding bars (full bars represent Apache MRQL query times). As can be observed, adding nodes to the cluster proportionally lowers the query time.

We next tested the scale-up characteristics of Apache VXQuery. We started by using a dataset that fits in the memory of each node (i.e., 7.2GB of data per node). The results appear in Figure 7.8 (again, Apache VXQuery query times correspond to the circles). While nodes and data are added to the query, the query time remains comparable, that is, the additional data is processed in the same amount of time. Apache VXQuery thus scales up well for Queries 7.2, 7.4, 7.6, and 7.8 on XML data.

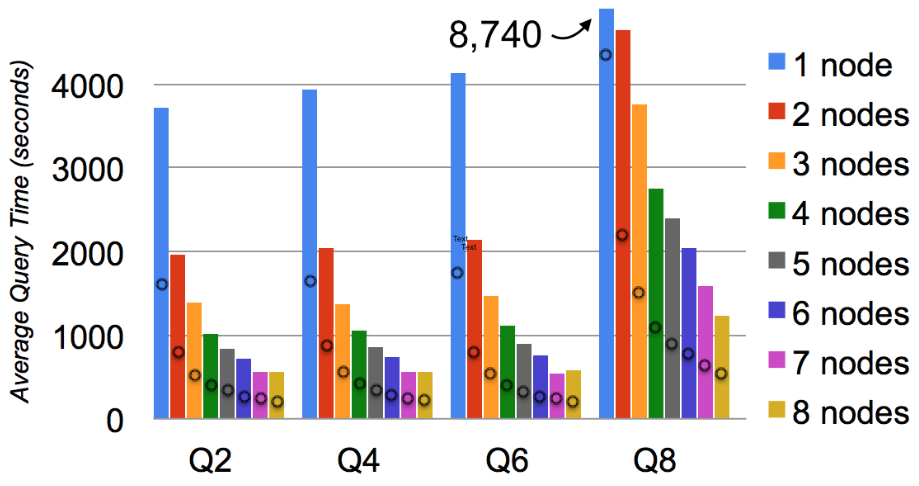


Figure 7.7: Apache VXQuery and Apache MRQL cluster speed-up (57GB dataset); circles mark the respective Apache VXQuery times.

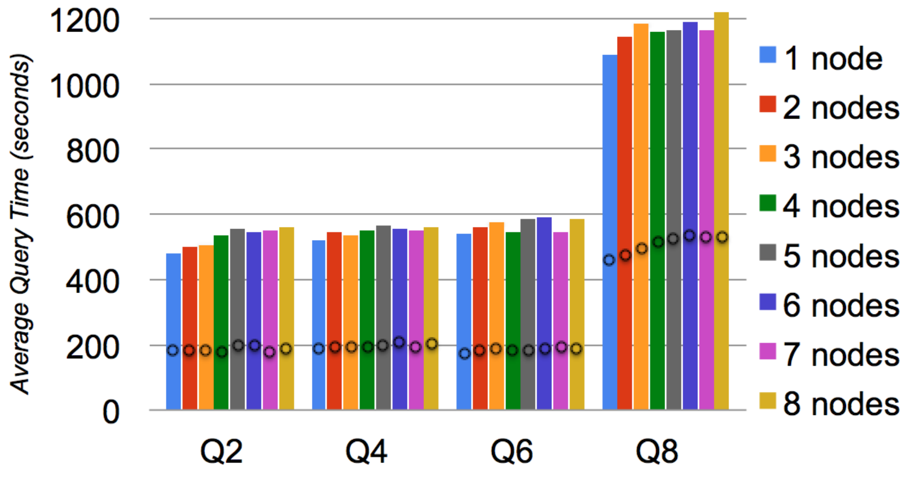


Figure 7.8: Apache VXQuery and Apache MRQL cluster scale-up (7.2GB per node); circles mark the respective Apache VXQuery times.

The next scale-up test utilizes all 528GB of weather data; here each node has 66GB of data split evenly on two local disks. The results appear in Figure 7.9; Apache VXQuery clearly scales-up well even for very large XML datasets.

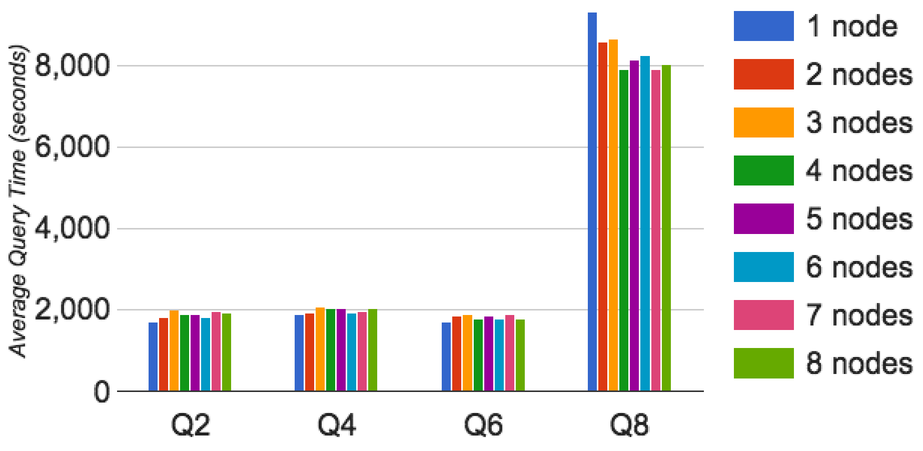


Figure 7.9: Apache VXQuery cluster scale-up (66GB per node).

Our final experiment sought to evaluate Apache VXQuery's performance against other open-source parallel XML processors. Among them, we chose Apache MRQL [27] as it was readily available. Using the HCN (57GB) dataset, we ran speed-up and scale-up tests for the same queries on Apache MRQL running on top of Hadoop 1.2.1 using MapReduce. Hadoop was configured with a 128MB block size and a replication factor of 1 (to reduce space on the cluster). Apache VXQuery outperforms Apache MRQL on all queries in terms of both scale-up and speed-up (Figures 7.7 and 7.8). Apache VXQuery's performance advantage comes partly from reading and parsing XML about two times faster than Apache MRQL. In addition, its richer set of operators provides for better performance. For example, VXQuery utilizes a Hybrid Hash Join algorithm that can keep a partition in main memory. Being MapReduce-based, Apache MRQL divides the join responsibility: partitioning is done by the mapper while the reducer joins the individual partitions. These two steps do not share state, yielding a traditional Grace Hash Join. On average over all experiments, VXQuery is 2.5x faster than Apache MRQL on Hadoop, validating the fact that building XQuery on top of a dataflow environment like Hyracks provides more opportunities for optimization and parallelism.

## 7.6 Conclusions

Apache VXQuery is a scalable open-source XQuery processor that we have built on top of Hyracks and Algebricks. We have described its implementation, including the XML data model dependent rewrite rules. These rules facilitate existing, data model independent Algebricks optimizations that serve to create efficient and parallel Hyracks jobs. We have demonstrated using a real 500GB dataset that VXQuery can scale out to the number of nodes available on a cluster for various XML selection, aggregation, and join queries.

Comparatively, Apache VXQuery is about 3.5x faster than Saxon on a single node and around 2.5x faster than Apache MRQL on a cluster in terms of scale-up and speed-up. The VXQuery source code is available at the Apache Software Foundation [8]; the current release contains approximately 100K LOC. We plan to add XQuery 3.0 features that support the analysis of Big Data, such as the *group by* and *window* clauses and to utilize indexing for increased query performance. Apache VXQuery developers are also adding support for large XML documents stored on a distributed file system and further optimizing the query compiler.

# Chapter 8

# Conclusions

In this dissertation, we looked at two types of data: data in the form of intervals and semistructured data stored as XML documents. Starting with interval data in Chapter 3, we examined five *overlapping interval join* algorithms. These state-of-the-art interval join algorithms were implemented in a real big data management system, Apache AsterixDB. This allowed us to check the algorithm performance under various scenarios within a memory budget. A series of experiments demonstrated that the algorithms showed good speed-up and scale-up performance on a single node and multi-node cluster (with a few exceptions). Different algorithms had the best performance on various scenarios, based on the characteristics of the joined relations. In Chapter 4, we describe CPU and I/O cost models for each interval join algorithm which can be used in a cost-based query optimizer to select the appropriate interval join algorithm. During algorithm implementation, the cost model either confirmed that the algorithm matched the expected efficiency, or helped us identify and correct inefficiencies with the implementation. In Chapter 5 we discussed our experiences based on the lessons learned from the implementation and testing of all algorithms. To experiment with all algorithms on Apache AsterixDB, we had to perform various changes to the system. These include support for selecting an interval join in the query language, a query compiler that creates an interval join query plan, data partitioning for interval data, and an efficient interval join operator. Among all algorithms, Time-Sweep Interval Join (TS) showed the most robust overall performance. The performance of Forward-Scan Interval Join (FS) and Sort-Merge Interval Join (SM) was very close (and SM is the simplest algorithm to implement.) Further, Overlap Interval Partition Join (OIP) showed good performance as long as the record size is small for limited memory, but the performance drastically deteriorated for large records. In our experimental setting, Disjoint Interval Partitioning (DIP) was the first one of the algorithms to spill. It is important for an optimizer to pick an algorithm that does not spill, since when spilling the performance slows down due to disk accesses. Based on the above, until a cost-based optimizer is available in Apache AsterixDB, we plan to implement TS and (for its simplicity and in case ordering is needed, SM). More algorithm choices can be added when the cost-based query optimizer is available.

Chapter 6 extends the work on overlapping interval joins to include Allen's relations, *covers* and *overlaps* (and their inverse, *covered-by* and *overlapped-by*). We presented detailed changes on how to modify the Sort-Merge Interval Join (SM) algorithm so as to support Allen's relations. The other algorithms can be similarly modified. A series of speed-up and scale-up experiments demonstrated good performance for modified SM algorithms on the *covers* and *overlaps* join conditions. A modified version of the TS algorithm that supports Allen's relations will also be available on the Apache AsterixDB project website [4].

Next we looked at creating scalable queries using XQuery on XML data in Chapter 7. As a result, we created Apache VXQuery which is a scalable open-source XQuery processor built on top of Hyracks and Algebricks. We have demonstrated using a real 500GB dataset that VXQuery can scale out to the number of nodes available on a cluster for various XML selection, aggregation, and join queries. Comparatively, Apache VXQuery is about 3.5x faster than Saxon on a single node and around 2.5x faster than Apache MRQL on a cluster in terms of scale-up and speed-up. The VXQuery source code is available at the Apache Software Foundation [8]; the current release contains approximately 100K LOC. Apache VXQuery developers are continuing to add additional support for things like large XML documents stored on a distributed file system.

# Bibliography

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, MatthiasJ. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, pages 1–26, 2014.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[4] Apache AsterixDB. `http://asterixdb.apache.org/`.

[5] Apache Flink. `http://flink.incubator.apache.org/`.

[6] Apache Hadoop. `http://hadoop.apache.org/`.

[7] Apache MRQL. `http://mrql.incubator.apache.org/`.

[8] Apache VXQuery. `http://vxquery.apache.org/`.

[9] Shivnath Babu and Herodotos Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[10] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[11] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. *ACM SIGMOD*, pages 479–490, 2006.

[12] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *IEEE ICDE*, pages 1151–1162, 2011.

[13] Vinayak Borkar, Yingyi Bu, Michael Carey, Nicola Onose, E. Preston Carman, Jr., and Till Westmann. Algebricks: A Framework for the Analysis and Efficient Evaluation of Data-Parallel Jobs. *ACM SOCC*, 2015.

[14] Vinayak Borkar and Michael J. Carey. A Common Compiler Framework for Big Data Languages: Motivation, Opportunities, and Benefits. *IEEE Data Eng. Bull.*, 36(1):56–64, 2013.

[15] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website*, 11:21, 2007.

[16] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *PVLDB*, 10(11):1346–1357, 2017.

[17] Francesco Cafagna and Michael H. Böhlen. Disjoint interval partitioning. *VLDB J.*, 26(3):447–466, 2017.

[18] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Paxquery: Efficient parallel processing of complex xquery. *IEEE TKDE*, 27(7):1977–1991, 2015.

[19] E. P. Carman, Jr., T. Westmann, V. R. Borkar, M. J. Carey, and V. J. Tsotras. Apache VXQuery: A Scalable XQuery Implementation. *CoRR*, abs/1504.00331, 2015.

[20] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing interval joins on map-reduce. In *EDBT*, pages 463–474, 2014.

[21] Hyebong Choi, Kyong-Ha Lee, Soo-Hyong Kim, Yoon-Joon Lee, and Bongki Moon. HadoopXML: A Suite for Parallel Processing of Massive XML Data with Multiple Twig Pattern Queries. In *CIKM*, pages 2737–2739, 2012.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[23] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT framework for logical XQuery optimization. *VLDB*, pages 168–179, 2004.

[24] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1459–1470, New York, NY, USA, 2014. ACM.

[25] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *ACM SIGMOD*, pages 683–694, 2004.

[26] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An Optimization Framework for Map-Reduce Queries. In *EDBT*, pages 26–37, 2012.

[27] Leonidas Fegaras, Chengkai Li, Upa Gupta, and Jijo Philip. XML Query Optimization in Map-Reduce. In *WebDB*, 2011.

[28] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax Experience. In *VLDB*, pages 1077–1080, 2003.

[29] Dengfeng Gao, S. Jensen, T. Snodgrass, and D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, March 2005.

[30] Hyracks. `https://code.google.com/p/hyracks/`.

[31] Interval Join Interactive Spill Graph. `https://www.desmos.com/calculator/z46d63dyqa`.

[32] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. What's in a crowd? analysis of face-to-face behavioral networks. *Journal of theoretical biology*, 271(1):166–180, 2011.

[33] E. Preston Carman Jr., Till Westmann, Vinayak R. Borkar, Michael J. Carey, and Vassilis J. Tsotras. A scalable parallel xquery processor. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 164–173, 2015.

[34] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *ACM SIGMOD*, pages 1173–1184, 2013.

[35] Michael Kay. SAXON: The XSLT and XQuery Processor, 2004.

[36] Michael Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.

[37] Shahan Khatchadourian, Mariano Consens, and Jérôme Siméon. ChuQL: Processing XML with XQuery Using Hadoop. In *CASCON*, pages 74–83, 2011.

[38] T. Y. Cliff Leung and Richard R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 383–394, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[39] John DC Little. A proof for the queuing formula: L= $\lambda$ w. *Operations research*, 9(3):383–387, 1961.

[40] Norman May, Sven Helmer, and Guido Moerkotte. Strategies for query unnesting in XML databases. *ACM TODS*, 31(3):968–1013, 2006.

[41] National Climate Data Center: Data Access. `http://www.ncdc.noaa.gov/data-access/`.

[42] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

[43] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.

[44] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Texts and Monographs in Computer Science. Springer, 1985.

[45] Christopher Ré, Jérôme Siméon, and Mary Fernández. A complete and efficient algebraic compiler for XQuery. *IEEE ICDE*, page 14, 2006.

[46] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[47] Using Oracle XQuery for Hadoop. `http://docs.oracle.com/cd/E53356_01/doc.30/e53067/oxh.htm`.

[48] XML Tutorial. `http://www.w3schools.com/xml/`.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.