UNIVERSITY OF CALIFORNIA, SAN DIEGO

GraphIVM: Accelerating Incremental View Maintenance through Non-relational Caching

A Thesis submitted in partial satisfaction of the

requirements for the degree

Master of Science

in

Computer Science

by

Gaurav Saxena

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor Alin Deutsch
Professor Victor Vianu

2015

The Thesis of Gaurav Saxena is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2015

DEDICATION

This work is dedicated to my wife Mugdha and my son Manasth, in addition to my parents from whom I borrowed last two years to engage in this endeavor. I wish to pay them back for this favor now.

EPIGRAPH

How do you eat an elephant?

One bite at a time

—Bill Hogan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Several individuals played a crucial part in this thesis. To begin with I would like to thank my advisor Yannis Papakonstantinou whom I came to know from a paper "Hypothetical Queries in OLAP Environment" in 2012 in the course of my professional work. Later, when I join his research group, he provided the initial motivation for this work and was instrumental in bringing it to fruition. I would also like to thank my collaborator Yannis Katsis who has always been my first stop for bouncing myriad of ideas, some of which have made it here.

In addition, I would also like to thank Kian Win Ong, for being the constructive critic of my work at every step. His keen insights on experiments and his unassailable authority on postgres helped shape crucial parts of this thesis. Furthermore, I would also like to thank my committee members Alin Deutsch and Victor Vianu for their feedback and guidance. Last, but not the least I would like to acknowledge my fellow students and all the professors who gave me a solid foundation to build this work upon.

This work is currently being prepared for publication. Parts of all the chapters will be included in the publication with co-authors Yannis Papakonstantinou and Yannis Katsis. The thesis author was the primary investigator and author of this material.

VITA

2005	Dual Degree (5-years Master). in Chemical Engineering, Indian Institute of Technology, Madras, India
	2005-2006 IBM India Private Limited, Gurgaon, India
2006-2009	Headstrong, Noida, India
	2009-2012 SCA Technologies, Gurgaon, India
	2012-2013 Times Internet Limited, Gurgaon, India
	2013-2014 Teaching Assistant and Graduate Student Researcher, University of California, San Diego
	2015 Master of Science, University of California, San Diego

ABSTRACT OF THE THESIS

GraphIVM: Accelerating Incremental View Maintenance through Non-relational Caching

by

Gaurav Saxena

Master of Science in Computer Science

University of California, San Diego, 2015

Professor Yannis Papakonstantinou, Chair

Incremental View Maintenance (IVM) is the process of incrementally maintaining the view when the underlying data change. Given the high frequency of data modifications in many practical scenarios, it is imperative that an IVM approach is as efficient as possible. One technique commonly used to accelerate IVM is the materialization of a set of additional auxiliary views, which can be leveraged to speedup the maintenance of the original view. However,

existing approaches assume that these auxiliary views are relational tables.

We argue that this assumption creates both space and time inefficiencies by introducing redundancies that would have been avoided if the auxiliary views were stored in a non-relational format. Based on this observation, we propose a novel non-relational auxiliary view, referred to as the *join graph*, and a corresponding *GraphIVM* system, which leverages the join graph to accelerate incremental view maintenance. The join graph, which intuitively represents how tuples of the underlying database join with each other, is shown to be compact and non-redundant, leading to an efficient IVM approach. This approach also benefits from two additional optimizations, described in the thesis, that allow it to further speedup the IVM process. Experiments of the GraphIVM system against state of the art IVM approaches verify that in all cases, but extremely simple views, GraphIVM significantly outperforms state of the art IVM approaches. More importantly, its speedup over other approaches increases as the views become more complex (measured in terms of fanout and number of joins).

Chapter 1

Introduction

Incremental View Maintenance (IVM) was originally introduced in the 80s [32, 5] to efficiently maintain materialized views when the underlying data change. The original motivating use case was keeping materialized views in data warehouses up to date. The following years saw a significant amount of research on IVM, leading to more efficient and/or general IVM approaches. The research interest in IVM subsequently declined as focus shifted away from data warehousing.

Recently however companies and researchers regained interest in IVM in the context of publish-subscribe architectures. Publish-subscribe platforms, such as social networking platforms (e.g., Twitter, Facebook or Linkedin), rely on technologies that enable the quick propagation of updates from the base tables (e.g., people's posts or comments) to query results (e.g, an addition of a

person's post has to be quickly propagated on the timeline's of his/her followers). These are essentially IVM technologies with the difference that the IVM system only needs to compute the updates to the view and not apply them to the latter, as the view is not maintained by the IVM system itself but by the client (in this case the social networking web-site).

More importantly, the publish-subscribe scenario places novel requirements on the performance characteristics of IVM approaches. In particular, updates have to be propagated from the base tables to the view as quickly as possible. For instance, twitter receives over 5700 tweets per second on average and has a record of 143,000 tweets per second [1]. Each of these tweets has to quickly reach several hundreds of users timeline on average [4] and thousands in the worst case [7].

An approach that has been historically used to accelerate IVM is materializing additional auxiliary views that are used to speed up the computation of view updates [15].

Example 1 *Consider three tables* $R(A, B)$*,* $S(B, C)$*, and* $T(C, D)$ *and a view* $V : R \bowtie S \bowtie T$ *computing their natural join. Now consider what happens when a new tuple* r *is added to base table* R*. As the result of this insertion the IVM system has to insert to the view all tuples computed through the following expression:* $\{r\} \bowtie S \bowtie T$ [1]*. Previous approaches observed that this process*

[1]The notation $\{r\}$ is used to denote a relational instance composed of a single tuple r

repeatedly (for every newly added tuple r) evaluates (part of) the relational expression $S \bowtie T$. To avoid this recomputation and accelerate the IVM process, many IVM approaches materialize this subexpression as a new auxiliary view $V_R : S \bowtie T$. Computing the tuples to be inserted to the view then reduces to computing $\{r\} \bowtie V_R$.

However, the set of auxiliary views that need to be generated to speedup IVM depends on the base tables that accept modifications.

Example 2 *Continuing our example, while $V_R : S \bowtie T$ is useful in computing view modifications resulting from changes to R, it is not useful for accelerating the maintenance of the view resulting from changes to relation T. Following similar reasoning to the one used to come up with the auxiliary view V_R it can be seen that the IVM of modifications to table T are best served instead by a second auxiliary view $V_T : R \bowtie S$.*

Accommodating multiple base table changes thus leads to the generation of multiple auxiliary views. This can very quickly lead to performance issues for two main reasons: First, the auxiliary views need to be themselves maintained whenever the base tables change. This view maintenance partially offsets the performance benefits of creating auxiliary views in the first place. Second, these auxiliary views typically contain redundant information, leading to a substantial storage overhead.

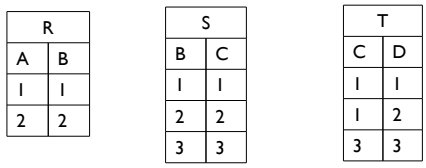
Example 3 *For instance, consider the views* $V_R : S \bowtie T$ *and* $V_T : R \bowtie S$ *introduced above to speed up the computation of view updates resulting from updates to base relations* R *and* T*, respectively. First, it is clear that when a tuple* r *of table* R *is modified, the IVM system has to maintain not only the original view* V *but also the auxiliary view* V_T*. Similarly, changes to table* S *will lead to maintenance of both* V_R *and* V_T*. This negatively affects the performance of the IVM approach. Second, the auxiliary views* V_R *and* V_T *contain in general redundant information, as they both contain a subset of relation* S*. In particular, any* s *tuple of* S *that joins with* R *and* T*, will appear in both views* V_r *and* V_t*.*

In this work, we argue that these inefficiencies of auxiliary views are a byproduct of the assumption made by prior work that these auxiliary views need to be relational tables. We show that by employing a single non-relational auxiliary view, we can speed up the view update computation for updates on *any* base table. This novel auxiliary view is guaranteed to be non-redundant, containing each base table tuple at most once. This not only reduces the memory footprint of the view compared to prior work, allowing it to scale to bigger data sets but also allows for more efficient maintenance of the auxiliary view. Our experiments (section 9.9) show that GraphIVM's memory requirement is several times lower than other approaches. In addition, it increases at a lower rate than the number of tuples for increasing join chain length and at similar rate with increasing fanout.

In a nutshell, this novel auxiliary view, referred to as the *join graph* is a graph capturing how tuples of the relations involved in the view definition join with each other. Leaving its formal presentation to chapter 5, we next explain it through a simple example.

Example 4 *Consider the instances of tables R, S and T shown in Figure 1.1a. Figure 1.1b shows the join graph for the view $V : R \bowtie S \bowtie T$. Ovals (which as we will see later) represent hyperedges correspond to tuples of the base tables and the intersection of two tuples show that the tuples join with each other (i.e., they contain the same values for the attributes of the corresponding relations that are used to join these two relations in the view). It is important to note that each base table tuple appears at most once (in this case it is exactly once), thus reduces the redundancy of the auxiliary view making it more efficient to update it. Finally, please note that the join graph contains not only the base table tuples that make it to the view, but even the tuples that join only with tuples of a subset of relations mentioned in the view. For instance, tuple $R(2, 2)$ appears in the join graph even though it only joins with an S tuple (i.e., tuple $S(2, 2)$) and not with a T tuple that would allow it to make it to the view. From that perspective, the join graph can be intuitively used to compute both auxiliary views V_R and V_T mentioned above without the associated redundancy.*

We present GraphIVM; an IVM approach that utilizes the join graph to accelerate incremental view maintenance and compare it to state of the art IVM

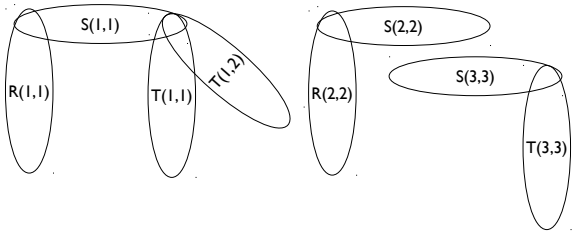


R	
A	B
1	1
2	2

S	
B	C
1	1
2	2
3	3

T	
C	D
1	1
1	2
3	3

(a) Database Instance



(b) Join Graph

Figure 1.1: Instance and corresponding join graph for view $V : R \bowtie S \bowtie T$ (Example 4)

approaches. Experimental results show that in most common cases GraphIVM significantly outperforms such IVM approaches.

Contributions. This work makes the following contributions:

- A novel graph-based, compressed, aggressively materialized auxiliary view that can be used to speed up IVM. In contrast to other sets of auxiliary views used in the IVM literature, this auxiliary view, referred to as the *join graph*, achieves non-redundancy by storing each base table tuple at most once. Additionally, the join graph contains enough information about the underlying base data to guarantee that it is both self-maintainable (i.e., it can be maintained without having to access the base tables) and it can be used to infer the view (so that there is no need to keep a separate copy

of the view).

- A novel IVM system, called GraphIVM, which leverages the concept of the join graph to incrementally maintain a view in an efficient way. GraphIVM has an architecture flexible enough to support both the data warehousing and publish-subscribe scenarios.

- A novel join-graph specific optimization, called *look-ahead* that further increases the performance of GraphIVM. To guarantee self-maintainability, the join graph in general contains not only base table tuples that join with each other but even base table tuples that are involved in partial joins (i.e., in joins that cannot be yet extended to an entire tuple that would appear in the view). However, when a base table tuple is modified, GraphIVM has to follow such partial join paths, to check whether as a result of this modification they may lead to new tuples in the view. The look-ahead optimization accelerates this process by storing information in the join graph, whether a join path is partial, so that GraphIVM avoids following it.

- A second optimization, called *view-specific compilation*, that allows GraphIVM to create view-specific maintenance code. Although this approach has been already employed by state-of-the-art IVM approaches, such as DBToaster [3], our work not only shows through experiments that this optimization

is applicable even in an IVM implementation based on an non-relational auxiliary view but even explores the performance gain of code compilation, comparing the GraphIVM system with and without this optimization.

- An experimental evaluation of GraphIVM against three alternative approaches: (a) traditional IVM without auxiliary views, (b) the DBToaster system [3], which corresponds to the state of the art in IVM (employing relational auxiliary views) and (c) an approach that keeps the same information as GraphIVM but in a relational form. The experimental results show that for everything but extremely simple queries GraphIVM outperforms all the above approaches and the performance difference increases as the queries become more complex.

Thesis Outline. This thesis is structured as follows: We start by describing related work in Chapter 2. We then proceed by describing two definitions of the IVM problem applicable to two different scenarios (publish-subscribe and data warehousing) in Chapter 3. Chapter 4 outlines the architecture of GraphIVM, which enables it to support both scenarios. The join graph - the non-relational auxiliary view that stands behind GraphIVM's efficiency - is described in Chapter 5, the algorithms powering each of GraphIVM's modules in Chapter 6 and a list of optimizations that further improve the performance of the system in Chapter 7. Chapter 8 presents alternative IVM approaches and Chapter 9 describes the experimental results of comparing GraphIVM against

these alternatives. Finally, Chapters 10 and 11 present future work and conclude the thesis respectively.

Chapter 2

Related Work

Incremental View Maintenance (IVM) has been extensively studied over the past three decades, leading to a vast literature in the topic. Shimueli et. al. [32] studied the problem of quickly applying updates to a materialized view for acyclic database for Project-Join queries and introduced the algebraic differencing technique. This was extended to SPJ class of views and multiple views by Blakely et.al. [5]. The class of queries was further extended to include outer join views [12] and recursive views [14]. Furthermore, Gouzhu et. al. have discussed the issue of maintaining views which cannot be described in relational calculus or SQL (like transitive closure) but can be maintained by these query languages in a detailed survey [9]

We next describe in detail the IVM aspects encountered in prior work that are closely related to our work on GraphIVM. For comprehensive surveys

on other aspects of IVM, the reader is referred to [13, 8].

Auxiliary Views in Query Answering. Auxiliary views have been used in many settings in the database literature. First and foremost, they have been used to accelerate query answering. By materializing an appropriate view, one can improve the performance of a query or set of queries. Researchers have looked not only at the problem of using such views when possible (a problem known as *answering queries using views*)[18, 22], but also at the problem of selecting which views to materialize. This problem, known as the *view selection problem* [2, 15, 16, 17, 33, 30], usually appears with varying formulations based on what is being optimized for. Common optimization goals in view selection include among others reducing the query response time [33], limiting the storage requirements [15] or reducing the view maintenance time [30, 16]. Apart from general relational views, prior work in query optimization has also looked at the problem of creating more specialized views that can be used by the query optimizers to optimize query execution. A prime example of such a view are join indices, used to optimize join evaluation [34]. The join index gave rise to the idea of the full outer join auxiliary view, which we consider as an alternative to GraphIVM and discuss it in Section 8.3.

Auxiliary Views in IVM. At a high-level of abstraction, any IVM approach involves some form of query answering (in particular, a query which operates on the changes that were performed on the base tables to produce the

changes that have to be applied to the view). As such, auxiliary view materialization has also been applied to IVM approaches. IVM approaches typically employ auxiliary views in two different ways: they either use operator-specific views to accelerate the maintenance of particular relational algebra operators (such as aggregation [27, 26] or top-k [37] operators) or they use general views that are exploited holistically during the maintenance of the auxiliary view [31, 24, 28, 3]. The join graph employed by GraphIVM falls under the second category, as it is used to accelerate the maintenance of the entire view, which may include among others join, projection and aggregation operators. However, in contrast to previous approaches, to the best of our knowledge GraphIVM is the first IVM approach to use a holistic *non-relational* view to accelerate IVM.

Query Compilation. Apart from views, another optimization that has been recently suggested in the context of IVM is the generation of a view-specific code for IVM. The idea of compiling a procedure into code has been widely explored in the query execution literature. Many works explored the idea of compiling query plans into code in order to speed up query evaluation. This led to approaches that among others proposed the execution of queries through iterative programs [10, 11], the conversion of SQL queries into query-specific JVM code [29] and the translation of queries to native code that eliminates the pipelining inherent in the DBMS-based query execution in favor of materializing results [21]. This idea was applied to the IVM problem by the DBToaster [3]

system, which suggested the generation of view-specific maintenance code. In this work we employ this optimization to further increase the performance of GraphIVM, as we will discuss in chapter 7.

Chapter 3

Problem Definition

We next formally define the IVM problem. We start by defining the diffs, which capture changes to the tables and then continue by defining the IVM problem and the supported view definition language.

In this work we consider maintenance of relational views. We use capital letters to represent relations (e.g., R, S, T) or attributes (e.g., A, B, C) with the difference made clear from the context, small letters to represent attribute values (e.g., a, b, c) and $\langle a, b, c \rangle$ to represent a tuple with attributes a, b, and c.

3.1 Diffs

Following the convention used by prior IVM works, modifications to a relation (be it a base relation or a view) are represented through diffs. Let R be a relation with attributes $A_1, A_2, \ldots, A_n$. A diff δ_R for relation R is a tuple

representing the insertion, deletion or update of a single tuple in R and they are respectively called *insert*, *delete* and *update diffs*. We next define these three types of diffs:

Definition 1 Insert/Delete Diff. *An* insert/delete diff δ_R^+/δ_R^- *for relation* $R(A_1, A_2, \ldots, A_n)$ *is a tuple* $\delta_R^+/\delta_R^- = \langle a_1, a_2, \ldots, a_n \rangle$ *representing the insertion/deletion of the tuple* $\langle a_1, a_2, \ldots, a_n \rangle$ *to/from* R.

Definition 2 Update Diff. *An* update diff δ_R^u *for relation* $R(A_1, A_2, \ldots, A_n)$ *is a tuple* $\delta_R^u = \left\langle a_1^{pre}, a_2^{pre}, \ldots, a_n^{pre}; a_1^{post}, a_2^{post}, \ldots, a_n^{post} \right\rangle$ *representing the update of the tuple* $\langle a_1^{pre}, a_2^{pre}, \ldots, a_n^{pre} \rangle$ *of* R *to* $\left\langle a_1^{post}, a_2^{post}, \ldots, a_n^{post} \right\rangle$. *The values* $a_1^{pre}, a_2^{pre}, \ldots, a_n^{pre}$ *correspond to the values in the tuple before the modification and are thus called pre-state values. Similarly, we refer to* $a_1^{post}, a_2^{post}, \ldots, a_n^{post}$ *as the post-state values.*

Let I be an instance of R and δ_R a diff for R. We denote by $\delta_R(I)$ the instance of R generated by applying the modification represented by δ_R on the relational instance I.

3.2 Problem Definition

As it has been implemented in data warehousing scenarios, incremental view maintenance takes as input an insert/delete/update diff on the base table, computes the corresponding view diffs that reflect the base table change and

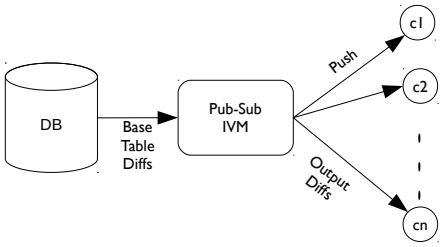
finally applies them to the view. The updated view can then be accessed by client queries as any other relation.

However, as discussed in the introduction, recently incremental view maintenance has been revisited in the context of publish-subscribe systems. In such systems, the view is materialized not in the database management system (DBMS) but instead in the application layer (e.g., by the code of a social networking web-site). Since the view maintenance is not done in the DBMS, the IVM system simply needs to transform the base table diffs to view diffs, which are then propagated to the client for processing.

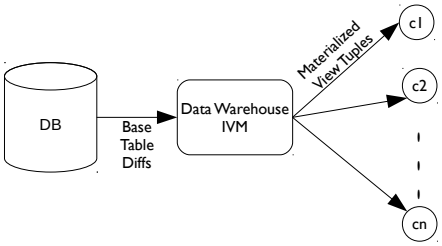
We next formally define the IVM problem for each of these two scenarios. Figures 3.1a and 3.1b graphically depicts the resulting definitions. Although these definitions are very similar (differing essentially in whether the diffs are applied to the view or not), as we will see in the experiments (Chapter 9), different IVM approaches exhibit in general different performance characteristics in each of the two scenarios.

3.2.1 Publish-Subscribe IVM

The structure of the IVM problem for publish-subscribe systems is shown in Figure 3.1a. As shown in the Figure, in this scenario the clients maintain an instance (or sub-instance) of the view and *subscribe* to changes in that view. The IVM system then computes the changes in the view as view diffs and



(a) Publish-Subscribe IVM



(b) Data Warehousing IVM

Figure 3.1: Publish-Subscribe and Data Warehousing IVM

pushes them to the client. This use case is typically found in systems handling maintenance of web views [19], real-time monitoring and continuous queries [6, 23]. We next formally define the IVM problem for such systems:

Definition 3 Publish-Subscribe IVM. *Let D be a database with relations $R_1, R_2, \ldots, R_n$, I a corresponding database instance and V a view over these relations. Consider also an input diff δ_{R_i} on some base relation R_i. IVM in publish-subscribe systems is the problem of computing a set of view diffs $\bar{\delta}_V = \{\delta_V^1, \delta_V^2, \ldots, \delta_V^m\}$, such that applying these diffs on the view has the same effect as applying the input diff to the base table and then computing the view (i.e., $\bar{\delta}_V(V(I)) = V(\delta_{R_i}(I))$).*

3.2.2 IVM in Data Warehousing

In contrast to the publish-subscribe systems, where clients consume view diffs produced by the IVM system, in data warehousing the IVM system also applies these diffs to the view. The clients can then access the updated view as any other database relation. Figure 3.1b shows the resulting IVM system. The IVM problem in this case is defined as follows:

Definition 4 Data Warehousing IVM. *Let D be a database with relations $R_1, R_2, \ldots, R_n$, I a corresponding database instance and V a view over these relations. Consider also an input diff δ_{R_i} on some base relations R_i. IVM in the data warehousing-subscribe is the problem of (a) similarly to IVM in the publish-subscribe scenario computing a set of view diffs $\bar{\delta}_V = \{\delta_V^1, \delta_V^2, \ldots, \delta_V^m\}$ such that applying the output diffs on the view has the same effect as applying the input diff to the base table and then computing the view (i.e., $\bar{\delta}_V(V(I)) = V(\delta_{R_i}(I))$) and (b) applying those diffs on the view instance $V(I)$.*

3.3 View Definition Language

In this work we consider views that are formulated from a query language containing most common Select-Project-Join queries with Aggregation. This language, referred to as $SPJA_p$, contains acyclic queries which can be expressed using a relational algebra plan containing:

- **Projection** operators under bag semantics.
- **Selection** operators with arbitrary conditions (including equalities, inequalities and functions), s.t. a selection operator appears in the plan directly over the scan of a base relation (which guarantees that each selection operator operates on attributes coming from a single base table).
- **Join** operators with equality conditions.
- **Aggregation** operators through associative aggregate functions operating over attributes of a single base table. Associative functions are those which can be incrementally maintained using only their previous value and the attribute values in the base table diffs. Examples of associative functions include among others COUNT, SUM, and AVERAGE.

Chapter 4

GraphIVM Architecture

GraphIVM supports both the publish-subscribe and data warehousing scenarios through the system architecture shown in Figure 4.1. Rounded boxes correspond to modules of the system, while rectangles represent internal data structures. GraphIVM employs a single internal data structure, denoted as the *join graph*. We next explain the different modules used by GraphIVM in detail:

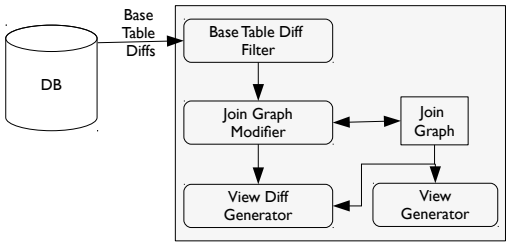


Figure 4.1: GraphIVM Architecture

Base Table Diff Filter. The GraphIVM system takes as input diffs representing modifications to the base tables [1]. As a first step it filters out all insert and delete diffs, which do not satisfy at least one of the selection conditions of the view, as these diffs do not lead to changes in the view. Update diffs are slightly more involved, since they contain both the old and new values of a tuple and depending on whether these values satisfy or not the selection conditions they may have to be converted to insert and delete diffs. We explain this case and give a detailed description of the base table diff filtering algorithm in Section 6.1. Note that this first filtering step is only possible because the selection operators in $SPJA_p$ views can be pushed all the way down to the base relations (and thus the selection conditions can be evaluated by looking only at a single base table tuple).

Join Graph Maintainer. Diffs that made it through the base table diff filter are given as input to the module responsible for maintaining the auxiliary data structure employed by GraphIVM, known as the join graph. The structure of the join graph is explained in Chapter 5 and the algorithm for maintaining it on incoming diffs in Section 6.2.

[1] Base table diffs can be computed in different ways. The most common ways are database triggers and modification logs. GraphIVM only requires base table diffs as input and is agnostic of the specific method used to generate them. For more information on base table diff generation, the interested reader can refer to previous work [20, 35].

The next two modules depend on the scenario in which GraphIVM is applied.

View Diff Generator. In the publish-subscribe scenario, GraphIVM employs the view diff generation module to create the view diffs that will be pushed to the clients. View diffs are created by traversing the join graph, as we will explain in Section 6.3.

Materialized View Generator. In the data warehousing scenario on the other hand, GraphIVM needs to also maintain the view. However, as we explained earlier, this view can be inferred from the - already updated - join graph and thus GraphIVM refrains from maintaining it in a separate data structure. Instead, whenever a client asks for the materialized view instance, GraphIVM invokes the materialized view generation module, which traverses the join graph to generate a relational representation of the view. The corresponding view generation algorithm is described in Section 6.4.

Chapter 5

Join Graph

In this Chapter we define the *join graph*; the auxiliary data structure used by GraphIVM to maintain the original view. How the join graph is used to maintain the original view and how it is itself maintained is discussed in Chapter 6.

To explain the presented notions we use a social networking example inspired by Twitter as described next.

Example 5 *Consider the schema of Figure 5.1a, containing four relations storing information about users and their followers, tweets and retweets. Consider also the view RetweetTracker of Figure 5.1b tracking for each tweet that was retweeted, who was the user that officially posted it and who retweeted it. We will next explain the join graph by referring to a sample instance of this database and to the corresponding view instance shown in Figure 5.2a and 5.2b, respec-*

```
User(username, gender, country)
Follower(username, followername)
Tweet(username, tweet, date)
Retweet(retweeter, retweet, name, tweet)
```

(a) Schema

```
CREATE VIEW RetweetTracker AS
SELECT U.username, U.gender, U.country,
       T.tweet, T.date, R.retweeter
FROM User U, Tweet T, Retweet R
WHERE U.username = T.username
      AND T.tweet = R.tweet
```

(b) View definition

Figure 5.1: Schema and view definition for running example

tively.

Although the join graph supports the maintenance of any $SPJA_p$view, for ease of exposition we present it in two stages: In the first stage we will present the subset of the join graph that is used to maintain SPJ views and then show how it can be extended to support arbitrary $SPJA_p$views.

5.1 SPJ Views

As the name suggests, the join graph is a compact graph-based representation of how tuples of relations involved in the view definition join with each other. Intuitively, one can think of the join graph as a hypergraph whose

User		
username	gender	country
joe	male	USA
bob	male	USA
alice	female	USA
cathy	male	Canada

Follower	
username	followername
joe	bob
joe	alice
cathy	bob
joe	cathy
bob	alice
cathy	alice

Retweet (R2)		
retweet	username	tweet
t2	bob	t1
t3	bob	t1
t4	alice	t1
t6	bob	t3
t7	alice	t3
t8	alice	t6

Tweet		
username	tweet	date
joe	t1	2/2/15
joe	t9	2/3/15

(a) Database instance

Retweet Tracker					
username	gender	country	tweet	date	retweeter
joe	male	USA	t1	1/12015	alice
joe	male	USA	t1	2/2/2015	bob
joe	male	USA	t1	2/2/2015	bob

(b) View instance

Figure 5.2: Database and view instance for running example

vertices are tuple attribute values and whose hyperedges contain the vertices corresponding to the attribute values contained in a base table tuple. Two hyperedges of different relations intersect when the corresponding base table tuples join with each other (i.e., they share the same value for the join attributes). As we will see though, to achieve compactness the actual join graph is slightly more involved.

Capturing join attribute values through join tuples. Instead of creating a distinct hyperedge for each base table tuple, the join graph further

compresses the data by creating a single hyperedge for all tuples of the same relation that share the same join attribute values. This summarization is based on the observation that all such tuples behave in the same way w.r.t. joins. Since a hyperedge summarizes many base table tuples based on their behavior w.r.t. joins, it is referred to as a *join tuple*.

Capturing projected attribute values through projected tuples. However, join tuples only capture the values of the join attributes of base table tuples. To be able to maintain the view, one needs to also capture the values of the attributes that are projected by the view. To this end, join tuples are annotated with a set of *projected tuples*, each containing the value of the projected attributes of one base table tuple. Thus the combination of a join tuple and the set of attached projected tuples capture all information of interest for all base table tuples that share the same values for the join attributes. The values of non-projected and non-joined attributes are ignored, as they are not of interest in maintaining the view.

Capturing join attributes through vertices. Finally, in order to further compress the representation, join tuples of different tables that join with each other do not contain distinct copies of the shared join attribute values. Instead they share the same object representing this join attribute values. This

object is called a *vertex value* and corresponds to a node in the join graph.

We next formally define all the components of a join graph in a bottom-up fashion.

5.1.1 Vertices and Vertex Values

Since the join graph represents how base table tuples join with each other, the attributes on which different tables join with each other are central for the approach. This concept of join attributes is formalized through the concept of *vertex* defined below. For ease of exposition our subsequent discussion assumes that each relation R appears only once in the FROM clause of the query. However, all definitions can be easily extended to multiple aliases for a single relation.

Definition 5 ***Vertex.*** *Let V be a view and R, S two relations in V, which join in V with each other on attributes $\bar{A} = \{A_1, A_2, \ldots, A_n\}$ and $\bar{B} = \{B_1, B_2, \ldots, B_n\}$ (i.e., V contains the join condition $R.A_i = S.B_i, \forall i \in \{1, 2, \ldots, n\}$). Then the $\langle \bar{A}, \bar{B} \rangle$ containing the ordered list of join attributes of relation R and S as its first and second component respectively constitutes a* vertex *for the join of R and S and is denoted by $J_{R,S}$. Given a vertex $J_{R,S} = \langle \bar{A}, \bar{B} \rangle$ we will use the notation $J_{R,S}.R$ (respectively $J_{R,S}.S$) to represent the list $\bar{A}$ (resp., $\bar{B}$) of join attributes of R (resp., S) in the join of R and S.*

In the following discussion we assume that for each view join between relations R and S, we consider only one vertex, which could be either $J_{R,S}$ or $J_{S,R}$ (the choice does not affect the following definitions).

Example 6 *Consider the view definition of Figure 5.1b. Since the relations User and Tweet join on attribute username, the join of these relations will be represented by the vertex* $J_{User,Tweet} = \langle\{User.username\}, \{Tweet.username\}\rangle$.

A particular instantiation of a vertex with values is called a *vertex value*. A vertex value, which intuitively corresponds to a set of values for the set of join attributes included in the vertex, is formally defined below:

Definition 6 ***Vertex Value.*** *Given a vertex* $J_{R,S} = \langle\{A_1, A_2, \ldots, A_n\}, \{B_1, B_2, \ldots, B_n\}\rangle$ *for the join of relations R and S, a* value for vertex *for* $J_{R,S}$ *is an ordered list of attribute values* $\bar{a} = \{a_1, a_2, \ldots, a_n\}$*, such that each value* a_i *is a valid value for attribute* $\bar{A}_i$ *of R and attribute* $\bar{B}_i$ *of S.*

Given a tuple t of relation R (resp. S), the vertex value $vv(t, J_{R,S})$ corresponding to t for vertex $J_{R,S}$ is the ordered list of values $\{a_1, a_2, \ldots, a_n\}$, s.t. a_i $(i = 1, 2, \ldots n)$ is t's value for attribute A_i.

Example 7 *Continuing our example, consider the database instance shown in Figure 5.2b. Let us focus for now on the first tuple of the User relation, which we will denote by* $r_1 = User(\langle joe, male, USA\rangle)$. *The vertex value for vertex*

$J_{User,Tweet}$ corresponding to tuple r_1 is $vv(r_1, J_{User,Tweet}) = \{joe\}$, since the value of tuple r_1 for the attribute on which the $User$ and $Tweet$ relations join is "joe".

A vertex value is central to the join graph's compactness. As we will see later , given an instance of a view V containing a join of relations R and S, for the entire set of tuples of both R and S that join with each other (i.e., they agree on the join attribute values), the corresponding join graph will contain a single vertex value. This comes in contrast to a relational representation of $R \bowtie S$, which would repeat the join attribute values.

5.1.2 Join Tuples

The join graph's compactness is further achieved through the notion of join tuples. As discussed above, a join tuple inferred from a base table tuple t of relation R intuitively contains only the values of the join attributes of t and it is used to represent all tuples of R that share the same join attribute values. Since join attribute values are encoded as vertex values, a join tuple is a set of vertex values. Formally:

Definition 7 ***Join Tuple.*** *Let V be a view, R a relation in V and $S_1, S_2, \ldots, S_m$ the set of relations R joins with in V. Then a join tuple of R is a tuple $\langle j_1, j_2, \ldots, j_m \rangle$, where $j_i, i = 1, \ldots, m$ is a vertex value for the vertex J_{R,S_i}.*

Given a tuple t of relation R, the corresponding join tuple $jt(t)$ is the tuple $\langle j_1, j_2, \ldots, j_m \rangle$, where j_i $(i = 1, 2, \ldots, n)$ is the vertex value $vv(t, J_{R,S_i})$ corresponding to t for vertex J_{R,S_i}.

Example 8 *Continuing our example, recall that the view contains a join between User and Tweet on the username and another join between Tweet and Retweet on attribute tweet. Consider the first tuple* $r_1 = User(\langle joe, male, USA \rangle)$ *of relation* $User$. *The corresponding join tuple can be written as* $jt(r_1) = \langle vv(r_1, J_{User,Tweet}) \rangle = \langle \{joe\} \rangle$. *Similarly, the join tuple corresponding to the first Tweet tuple* $r_2 = Tweet(\langle joe, t1, 2/2/15 \rangle)$ *is*
$jt(r_1) = \langle vv(r_1, J_{User,Tweet}), vv(r_1, J_{Tweet,Retweet}) \rangle = \langle \{joe\}, \{t_1\} \rangle$.

5.1.3 Projected Tuples

As discussed earlier, join tuples succinctly represent the join attribute values of base table tuples. However, they do not capture the values of the base table tuples for the attributes of the base relation that are projected by the view. This is captured by the concept of projected tuples, defined below.

Definition 8 ***Projected Tuple.*** *Consider a view* V *and a relation* R *mentioned in* V. *Let* $\bar{P} = \{P_1, P_2, \ldots, P_n\}$ *be an ordered list of the set of attributes of* R *that appear in the projection list of* V. *Then a* projected tuple *for relation* R *is a tuple* $\langle p_1, p_2, \ldots, p_n \rangle$, *s.t. each* $p_i, i = 1, 2, \ldots, n$ *is a valid value*

for attribute P_i. Each projected tuple has also an associated natural number c, referred to as the count *of the projected tuple.*

Given a tuple t of relation R, the corresponding projected tuple $pt(t)$ is the tuple $\langle a_1, a_2, \ldots, a_n \rangle$, where a_i is t's value for attribute P_i. The projected tuple's count is in this case 1.

The count of a projected tuple signifies how many base table tuples that have the same projected attribute values are summarized by the particular projected tuple.

In the join graph, a projected tuple is always associated with a particular join tuple.

Example 9 *In our running example the attributes of relation $User$ that are projected in the view are username, gender and country. Therefore the projected tuple corresponding to the User tuple $r_1 = User(\langle joe, male, USA \rangle)$ is $pt(r_1) = \langle joe, male, USA \rangle$. Notice that the projected tuple includes username* joe *even though it is already included in the join tuple (as it is also an attribute involved in a join). Similarly, for the Tweet tuple $r_2 = Tweet(\langle joe, t1, 2/2/15 \rangle)$ the corresponding projected tuple is $pt(r_2) = \langle t1, 2/2/15 \rangle$.*

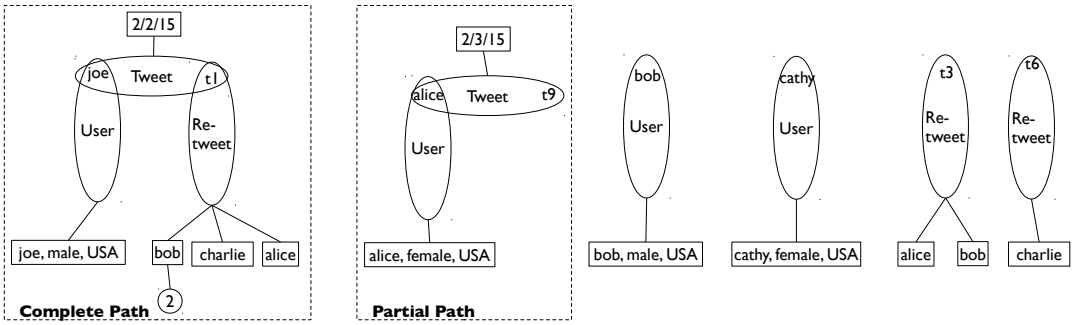


Figure 5.3: Join graph for our running example

5.1.4 Join Graph

A join graph uses a set of vertex values, join tuples created using these vertex values and projected tuples attached to these join tuples to represent a database instance. Formally:

Definition 9 ***Join Graph.*** *A* join graph $\langle d \rangle$ *consists of a set of vertex values* $\mathcal{VV}$*, a set of join tuples* $\mathcal{J}$ *containing vertex values in* $\mathcal{VV}$ *and a set of projected tuples* $\mathcal{P}$*, each attached to a join tuple in* $\mathcal{J}$*.*

For a given view and database instance, GraphIVM maintains the corresponding join graph. Consider a database D consisting of relations $R_1, R_2, \ldots, R_n$, a view V over D and a database instance I. The join graph for I w.r.t. V is a join graph $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$, s.t.:

- $\mathcal{VV}$ contains for each vertex J_{R_i,R_j} and for each set of tuples $\bar{t}$ of R_i (resp. R_j) that share the same values for attributes in $J_{R_i,R_j}.R_i$ (resp. $J_{R_i,R_j}.R_i$), a single vertex value $vv(t, J_{R_i,R_j})$, where t is an arbitrary tuple in $\bar{t}$.

- $\mathcal{J}$ contains for each set of tuples $\bar{t}$ of relation R_i $(i = 1, 2, \ldots, n)$ that share the same join attribute values a single join tuple $jt(t)$, where t is an arbitrary tuple in $\bar{t}$. The join tuples are constructed using vertex values in $\mathcal{VV}$.

- $\mathcal{P}$ contains for each set $\bar{t}$ of tuples of relation R_i $(i = 1, 2, \ldots, n)$ that share the same join attribute and projected attribute values a single projected tuple $pt(t)$ where t is an arbitrary tuple in $\bar{t}$. This projected tuple is attached to the tuple $jt(t) \in \mathcal{J}$ with count equal to the number of tuples in $\bar{t}$.

We next present an example of a join graph and explain the graphical notation that we use to represent join graphs.

Example 10 *Figure 5.3 shows the graphical representation of the join graph for the database instance of our running example (Figure fig:SchemaInstance) w.r.t. to the RetweetTracker view. Join tuples are graphically depicted as ovals. The relation corresponding to the join tuple is shown in the middle of the oval, while the vertex values contained in it are shown in its corners. For instance the leftmost oval graphically depicts a join tuple of relation User with vertex value* $\{joe\}$*. Similarly the oval with which this oval intersects corresponds to a join tuple of relation Tweet with two vertex values* $\{joe\}$ *and* $\{t1\}$*. Projected tuples are shown as rectangles connected to the join tuple to which they are attached. For instance, there is a single projected tuple* $\langle joe, male, USA\rangle$ *attached to the*

leftmost join tuple. If a projected tuple has count 1, then this is omitted from the graphical representation; otherwise it is shown in a circle connected to the projected tuple. For instance the projected tuple $\langle bob \rangle$ *has count of 2. Finally, by looking at the graphical representation of the join graph, it is obvious that through the concepts of join tuples, projected tuples and vertex values, it is a compact representation showing how the tuples of the underlying database instance join with each other.*

A central concept in the join graph is the concept of a path. Intuitively a path in a join graph corresponds to a set of join tuples of different relations, such that for each pair of join tuples in the set, one join tuple is "connected" to the other through other join tuples in the set. Formally:

Definition 10 ***Path.*** *Let* $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$ *be a join graph and* $P = \{j_1, j_2, \ldots, j_n\} \subseteq \mathcal{J}$ *a set of join tuples of this graph. This set is called a* path *iff the following two conditions hold:*

- *No join tuples in* P *correspond to the same relation.*
- *For every pair of join tuples* $k_1, k_m \in P$ *there exists a set* $\{k_1, k_2, \ldots, k_m\} \subseteq P$ *such that* k_i *and* k_{i+1} *(*$i = 1, \ldots, m-1$*) share a common vertex value.*

Example 11 *The set of three join tuples within the leftmost dashed box in Figure 5.3 form a path.*

We distinguish between two types of paths: a *complete path*, which includes join tuples of every relation mentioned in the view and a *partial path*, which does not include join tuples of at least one relation in the view.

Example 12 *Figure 5.3 contains two dashed boxes, denoting a complete and partial path. The path corresponding to the rightmost box is partial because it is missing a join tuple of relation Retweet, which is mentioned in the view.*

As we will see in Chapter 6, paths in a join graph form the underpinnings of the IVM algorithm used by GraphIVM.

5.2 SPJA$_p$ Views

If a view contains also aggregations, we could maintain it by keeping the join graph for the non-aggregated (i.e., SPJ) sub-expression of the view and then treat the aggregation as an additional post-processing step during view maintenance. However, this approach would in general introduce unnecessary overhead, as the join graph would be maintaining more information than is necessary for the efficient maintenance of the view.

To avoid this problem, we extend the join graph to keep information on aggregated values. In the following discussion we assume that it is possible to push aggregation operators in the relational algebra representation of the view down to single relations (if an aggregation cannot be pushed down or can be

pushed down only partially, the aggregation operator that cannot be pushed down to individual relations will be considered by the maintenance algorithm as additional post-processing that has to be carried out after the processing of the join graph).

Consider a view V mentioning relation R and an aggregate operator $\gamma_{\bar{G};\bar{f}}$, where $\bar{G}$ is a set of group by attributes and $\bar{f}$ a set of associative aggregation functions. Assume also that in the relational algebra representation of the view $\gamma_{\bar{G};\bar{f}}$ has been pushed down on top of relation R [1]. To create the join graph for an instance w.r.t. V, $\gamma_{\bar{G};\bar{f}}(R)$ is considered as a base relation. However, when it comes to creating projected tuples, instead of creating regular projected tuples, we will have to create *extended projected tuples* that capture also information on the aggregation. We next formally define the notion of the extended projected tuple.

Definition 11 Extended Projected Tuple. *Consider a view V and the operator $\gamma_{\bar{G};\bar{f}}$ pushed down to R in the relational algebra representation of V. Then an* extended projected tuple *for R is a tuple $\langle \bar{g}, \bar{a} \rangle$, where $\bar{g}$ are valid attribute values for the respective attributes in $\bar{G}$ and $\bar{a}$ are valid output values for the respective aggregation function $\bar{f}$. Each extended projected tuple has also an associated natural number c, referred to as the* count *of the extended projected tuple.*

[1] This approach can be easily extended to the case where a selection exists between the aggregate operator and the scan operator.

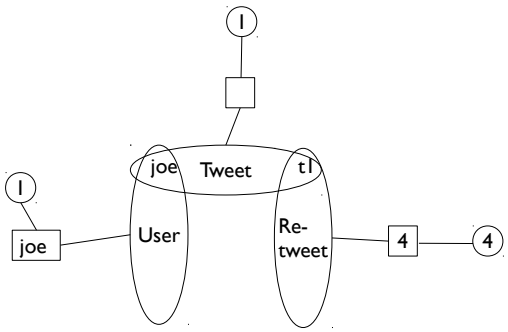


Figure 5.4: Hypergraph representation of the view RetweetCounter

The count of an extended projected tuples intuitively represents the number of base table tuples that were aggregated to produce the extended projected tuple. The following example illustrates this concept:

Example 13 *Consider the schema of our running example shown in Figure 5.1a and the following aggregated version TweetReachCounter of the view of our running example:*

```
SELECT U.username, count(R.tweet)
FROM User U, Tweet T, Retweet R
WHERE U.username = T.username
AND T.tweet = R.tweet
GROUP BY U.username
```

Figure 5.4 shows a part of the join graph corresponding to this view. The Retweet join tuple has an attached extended projected tuple that has the value 4 for the aggregate function COUNT and an associated count of 4 (since 4 base tuples were aggregated to produce this extended projected tuple).

Chapter 6

GraphIVM

Having defined the join graph, we can now describe the inner workings of each of the GraphIVM modules shown in Figure 4.1. As explained in Chapter 4, GraphIVM incrementally propagates base table diffs to a view V by following the following three major steps: (a) Filtering out base table diffs that do not satisfy the selection conditions of V, (b) applying the diffs that made it through the filtering step to the join graph, and (c) depending on the scenario (publish-subscribe vs datawarehousing) using the join graph to either create corresponding view diffs sent to the clients of the system or generating the entire materialized view when a client requests the latter. We next describe each of these steps.

6.1 Filtering Base Table Diffs

As described above , GraphIVM only reflects in the join graph those base table tuples that satisfy the selection conditions of the view. Thus, when a new base table diff δ_R for relation R arrives, GraphIVM first runs it through a filter to check whether it satisfies the selection conditions on R placed by the view. Depending on whether the diff satisfies the selection conditions, GraphIVM either propagates it to the next module or simply discards it.

Algorithm 1 shows the filtering procedure. We distinguish three cases, depending on the type of the base table diff. For our subsequent discussion, let V be a view mentioning relation R, δ_R a diff on R and S a set of selection conditions placed by V on R.

Filtering insert diffs. If δ_R is an insert diff (lines 1-4 of Algorithm 1) and it does not satisfy at least one of the selection conditions in S, then the modification represented by the diff does not need to be reflected in the join graph and thus the diff is simply discarded.

Filtering delete diffs. Similarly, if δ_R is a delete diff (lines 5-8) and does not satisfy some selection condition in S, then it is also discarded, as the diff represents the deletion of a tuple that does not satisfy the selection conditions and thus is not included in the join graph to begin with.

Filtering update diffs. The most interesting case is when δ_R is an update diff (lines 9-26). In that case, the diff may represent the update of a

tuple that was not in the join graph (because it did not satisfy some selection conditions before the update) but should be included now (because the updated values satisfy all selection conditions). If this is the case (lines 22-24), the update diff δ_R is converted to an insert diff by keeping only the post-state values of the initial update diff. Similarly an update diff may be converted to a delete diff (when the pre-state values satisfy the conditions but the post-state values do not, as shown in lines 19-21), simply propagated as an update diff (when both pre-state and post-state values satisfy the conditions, as shown in lines 17-18) or simply discarded (when both pre-state and post-state values do not satisfy the conditions, as shown in lines 25-26).

Note that this filtering procedure is only possible because each selection condition in in $SPJA_p$ views refers to attributes of a single relation at a time.

6.2 Maintaining the Join Graph

A diff produced by the base table diff filter corresponds to a modification that should be reflected in the join graph to bring it up to sync with the base data. This is accomplished by the join graph maintenance module, which uses three different maintenance algorithms depending on the type of the incoming diff (i.e., insert, delete or update).

The input to the join graph maintenance algorithms is a base table diff (δ_R^t) where t is the type of diff and R is the corresponding base table. Given

this input diff δ_R^t, the join graph maintenance algorithm modifies the join graph to reflect δ_R^t and also outputs information on what changes it made to the join graph. This information, denoted as *join graph modification* will be later consumed by the view diff generator algorithm to create the resulting view diffs.

Before describing the join graph maintenance algorithms, let us first define the set of possible join graph modifications, which these algorithms may produce:

Definition 12 Join Graph Modification. *A modification of join graph* $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$ *can be one of the following:*

- *Insert modification:* $ins(j, \bar{p})$*, representing the insertion of a join tuple* j *to* $\mathcal{J}$ *with attached projection tuples* $\bar{p}$*.*
- *Delete modification:* $del(j, \bar{p})$*, representing the deletion of the join tuple* $j \in \mathcal{J}$*, where* $\bar{p} \subseteq \mathcal{P}$ *is a subset of the projected tuples attached to* j*.*
- *Update modification:* $upd(j, p_{pre}, p_{post})$*, representing the update of the join tuple's* $j \in \mathcal{J}$ *projected value* p_{pre} *to* p_{post}*.*

We can now describe the join graph maintenance algorithms for each type of base table diffs. During the following discussion on the algorithms please note that all algorithms also maintain the "look-ahead"; an additional piece of information attached to join tuples to further improve the performance

of performing IVM. We discuss look-ahead in detail in the optimization section (Section 7.1).

6.2.1 Maintaining the join graph for insert diffs

An insert diff δ_R^+ for relation R may result in modifications of either projected tuples or join tuples in the join graph.

Let t be the newly inserted R tuple described by the diff. If the join tuple $jt(t)$ corresponding to t already exists in the join graph (i.e., t has the same join attribute values as another tuple of R represented in the join graph), then the algorithm has to either add a new projected tuple $pt(t)$ to the join graph and attach it to $jt(t)$ if $pt(t)$ does not already exist, or increment the count of $pt(t)$ by one if $p(t)$ already exists in the join graph.

On the other hand, if $jt(t)$ does not already exist in the join graph, $jt(t)$ together with $pt(t)$ have to be added to the join graph. The creation of $jt(t)$ may also lead to the generation of new vertex values in the graph. This resulting algorithm is Algorithm 2.

6.2.2 Maintaining the join graph for delete diffs

A delete diff represents the deletion of a base table tuple. The join graph maintenance algorithm reflects this deletion in the join graph by finding the corresponding projected tuple and reducing its count by one. However,

this deletion can lead to deletion of up to three types of objects in the graph; projected tuples, join tuples and vertex values.

If the count of the projected tuple becomes zero, then the projected tuple has to be removed from the join graph and from the join tuple to which it is attached. If this leads to the join tuple having zero attached projected tuples, then the join tuple has to be removed as well. Finally, if due to the deletion of the join tuple some vertex value ends up not being contained in any join tuple, this vertex value has to be also removed from the join graph. The resulting algorithm is Algorithm 3.

6.2.3 Maintaining the join graph for update diffs

An update base table diff δ_R^u may lead to changes to both join tuples and projected tuples, based on the attributes of the corresponding base table tuple that are updated. Algorithm 4 shows the procedure followed by GraphIVM to reflect an update diff in the join graph.

Let δ_R^u be an update diff for relation R describing the update of tuple t_{pre} of R to t_{post}. If the diff describes the update of values of projected attributes (lines 1-11 of the algorithm), then the system increases the count of the new projected tuple $pt(t_{post})$ by one and decreases the count of the old projected tuple $pt(t_{pre})$ by one. Special cases are handled as expected. For instance, if the new projected tuple does not already exist, it is created and if the old projected

tuple ends up having count equal to zero, it is removed.

If the diff describes the update of values of join tuples (lines 12-29), then GraphIVM moves the projected tuples attached to the old join tuple $jt(t_{pre})$ to the new join tuple $jt(t_{post})$.

6.3 Generating View Diffs

The join graph maintenance algorithm brings the join graph in sync with the base tables. Then, depending on the IVM scenario, GraphIVM activates different modules. In the case of a publish-subscribe scenario, GraphIVM should create the view diffs representing the modifications that have to be applied to the view and ship them to the clients that are listening for diffs. To this end, it invokes the view diff generation module, which creates the view diffs by using the join graph together with the information on which join and projected tuples were modified by the join graph maintenance algorithm. We next describe this algorithm, leaving the discussion of the module activated in the data warehousing scenario for the next section.

Given a modified join tuple j or projected tuple p of j in the join graph, the view tuples that have to be modified are all tuples corresponding to complete paths including j. To find all such paths, the view diff generation algorithm starts from the modified join tuple j and explores the entire graph by visiting

neighboring join tuples[1] in a Breadth-First-Search (BFS) fashion. Once a complete path is found, the algorithm appropriately combines the projected tuples of the join tuples included in the path to create entire view tuples that have to be inserted, deleted or updated in the view.

Algorithm 5 summarizes the resulting procedure, which is split into three subprocedures depending on the graph of join graph modification (insert, delete or update) that lead to invocation of the the view diff generation module.

6.4 Materializing the View

While in the publish-subscribe scenario GraphIVM has to produce view diffs, in the data warehousing scenario GraphIVM this is not required. In this case the IVM process is completed after the system maintains the join graph as explained in Section 6.2.

However, in this data warehousing scenario, a client may at some point in time request a copy of the materialized view in relational form. As we have discussed above, GraphIVM does not explicitly maintain the view in relational form. Instead it captures all information that is needed to recreate the view in the join graph and can recreate on demand such a relational representation of the view. The view intuitively corresponds to all tuples that can be created from complete paths in the join graph. Thus, in order to create such a view, it suffices

[1]Given a pair of join tuples j and j' of different relations, they are said to be *neighboring* tuples if and only if they share the same vertex value

to pick an arbitrary relation R mentioned in the view and run sub-routine *tupleGeneratorForInsDel* of Algorithm 5 for every join tuple of R. Algorithm 6 outlines the resulting procedure.

6.5 Indices

All algorithms presented above rely on the ability to quickly find a join tuple, projected tuple and vertex value. To ensure that this is the case, GraphIVM employs the following three indices; one for each of the above concepts:

- *Join tuple index:* Given a set of join attribute values, the index returns the join tuple that contains all these join attribute values.
- *Projected tuple index:* Given a join tuple j and an ordered list $\bar{a}$ of attribute values, the index returns the projected tuple p of j, which consists of values $\bar{a}$.
- *Vertex value index:* Given a vertex $J_{R,S}$ and an ordered list of attribute values $\bar{a}$, the index return the vertex value for $J_{R,S}$ that contains the values $\bar{a}$.

Data: (a) Base table diff δ_R for relation R,
(b) Set S of selection conditions on R
Result: Diff δ'_R for table R or nothing ($\emptyset$)

```
1  if δ_R is an insert diff (i.e., δ_R = δ_R^+) then
2      foreach selection condition s in S do
3          if values of δ_R^+ do not satisfy s then
4              return ∅
           end
       end
       return δ_R^+
   end
5  if δ_R is a delete diff (i.e., δ_R = δ_R^-) then
6      foreach selection condition s in S do
7          if values of δ_R^- do not satisfy s then
8              return ∅
           end
       end
       return δ_R^-
   end
9  if δ_R is an update diff (i.e., δ_R = δ_R^u) then
10     preSatisfySelect = true
11     postSatisfySelect = true
12     foreach selection condition s in S do
13         if pre-state values of δ_R^u do not satisfy s then
14             preSatisfySelect = false
           end
15         if post-state values of δ_R^u do not satisfy s then
16             postSatisfySelect = false
           end
       end
17     if preSatisfySelect and postSatisfySelect then
18         return δ_R^u
       end
19     if preSatisfySelect and not postSatisfySelect then
20         Convert δ_R^u to delete diff δ_R^- by keeping pre-state values only
21         return δ_R^-
       end
22     if not preSatisfySelect and postSatisfySelect then
23         Convert δ_R^u to insert diff δ_R^+ by keeping post-state values only
24         return δ_R^+
       end
25     if not preSatisfySelect and not postSatisfySelect then
26         return ∅
       end
   end
```

Algorithm 1: Base Table Diff Filtering

Data: (a) Insert base table diff δ_R^+ for relation R,
describing the insertion of tuple t
(b) Join graph $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$
Result: Set of join graph modifications

```
1  if jt(t) ∈ 𝒥 then
2      if pt(t) ∈ 𝒫 then
3          Increment count of pt(t) by 1
       else
4          Add pt(t) to 𝒫 with count 1
5          Add pt(t) to list of projected tuples in jt(t)
       end
6      if pt(t) contains an aggregated attribute then
7          Re-calculate aggregate value
       end
   else
8      Find or create, if absent, vertex values vv from δ_R^+
9      Add jt(t) to 𝒥
10     foreach vertex value v in vv do
11         if v ∉ 𝒱𝒱 then
12             Insert v to 𝒱𝒱
           end
13         Insert jt(t) to v
14         Update local look-ahead of all join tuples in v
15         if pt(t) contains an aggregated attribute then
16             calculate aggregate value
           end
       end
   end
17 return {ins(jt(t), {pt(t)})}
```

Algorithm 2: Join Graph Maintenance for Insert Diffs

Data: (a) Delete base table diff δ_R^- for relation R,
describing the deletion of tuple t
(b) Join graph $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$
Result: Set of join graph modifications

```
if pt(t) contains an aggregated value then
    Re-calculate Aggregated value
end
Reduce the count of pt(t) by 1
if count of instances of pt(t) = 0 then
    Remove pt(t) from jt(t)
    Remove pt(t) from P
end
if number of projected tuples in jt(t) = 0 then
    foreach vertex value v in vertex values of jt(t) do
        Remove jt(t) from v
        if v contains no join tuple then
            Remove v from VV
            Update local look-ahead of all join tuples in v
        end
    end
end
return {del(jt(t), {pt(t)})}
```

Algorithm 3: Join Graph Maintenance for Delete Diffs

Data: (a) Update base table diff δ_R^u for relation R,
describing the update of tuple t_{pre} to t_{post}
(b) Join graph $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$
Result: Set of join graph modifications

if δ_R^u *updates projected attributes* **then**
  Find new projected tuple $pt(t_{post})$
  if $pt(t_{post}) \in \mathcal{P}$ **then**
    Increment count of $pt(t_{post})$ by 1
  else
    Add $pt(t_{post})$ to $\mathcal{P}$ with count 1
    Add $pt(t_{post})$ to $jt(t_{pre})$
  end
  Find old projected tuple $pt(t_{pre})$
  if *count of* $pt(t_{pre}) > 1$ **then**
    Reduce count of $pt(t_{pre})$ by 1
  else
    Remove $pt(t_{pre})$ from $jt(t_{pre})$
    Remove $pt(t_{pre})$ from $\mathcal{P}$
  end
end
$\bar{p}$ = all projected tuples of $jt(t_{pre})$
if δ_R^u *updates join attributes* **then**
  Find old join tuple $jt(t_{pre})$
  Find old vertex values $oldVV$ from t_{pre}
  foreach *vertex value* $v \in oldVV$ **do**
    Remove $jt(t_{pre})$ from v
    if v *contains no join tuples* **then**
      remove v from $\mathcal{VV}$
    end
  end
  if *If* $jt(t_{post})$ *doesn't exists* **then**
    Add $jt(t_{post})$ to $\mathcal{J}$
    Update local look ahead for $jt(t_{post})$
  end
  copy p to $jt(t_{post})$
  Find or create new vertex values $newVV$ from $jt(t_{post})$
  foreach *vertex value* $v \in newVV$ **do**
    Add $jt(t_{post})$ to v
    if $v \notin \mathcal{VV}$ **then**
      Add v to $\mathcal{VV}$
    end
  end
  Remove $jt(t_{pre})$ from $\mathcal{J}$
  return $\{del(jt(t_{pre}), \bar{p})\}$, $\{ins(jt(t_{post}), \bar{p})\}$
else
  return $\{upd(jt(t), pt(t_{pre}), pt(t_{post}))\}$
end

Algorithm 4: Join Graph Maintenance for Update Diffs

```
Data: (a) Set of join graph modifications S
      (b) Join graph ⟨𝒱𝒱, 𝒥, 𝒫⟩
Result: Set of view diffs Δ
1  foreach s ∈ S do
2      if s is an insert join graph modification then
3          t = tupleGeneratorForInsDel(s(j), s(p̄))
4          Convert t to δ_V^+ and add to Δ
       end
5      if s is a delete join graph modification then
6          t = tupleGeneratorForInsDel(s(j), s(p̄))
7          Convert t to δ_V^- and add to Δ
       end
8      if s is a update join graph modification then
           t = tupleGeneratorForUpdates(s(j), s(p̄))
9          Convert t to δ_V^u and add to Δ
       end
10     return Δ
   end
   Procedure tupleGeneratorForInsDel(j, p̄, ⟨𝒱𝒱, 𝒥, 𝒫⟩)
11     Starting from j, visit join tuples in 𝒥 reachable from j in BFS fashion considering complete paths only
12     C be a relation with schema identical to the view
13     foreach complete path discovered by BFS do
14         Add to C, the cartesian product of p̄ and the projected tuples, for all the join tuples on path
           except j
       end
15     A = aggregator(C)
16     return A
   Procedure tupleGeneratorForUpdates(j, p_pre, p_post, ⟨𝒱𝒱, 𝒥, 𝒫⟩)
17     Starting from j, visit join tuples in 𝒥 reachable from j in BFS fashion considering complete paths only
18     C be a relation with schema identical to the view
19     foreach complete path discovered by BFS do
20         Add to C the cartesian product of the projected tuples, for all the join tuples on path except j
       end
21     A = aggregator(C)
22     A = t_pre × A; t_post × A
23     return A
   Procedure aggregator(C)
24     foreach tuple t in C do
           t_f = Multiply counts of projected tuples
25         if view is aggregated then
26             foreach aggregated value v in t do
27                 v = v × t_f
               end
28             A = group C on group by attributes and compute new aggregated values
           else
29             A = Repeat each tuple t in C t_f times
           end
       end
30     return A
```

Algorithm 5: View Diff Generation

Data: (a) $\langle j, p \rangle$, where j and p the modified join
and projected tuple, resp.
(b) Join graph $\langle \mathcal{VV}, \mathcal{J}, \mathcal{P} \rangle$

Result: View Instance as bag of tuples

1 Arbitrarily choose a relation R of the view
2 **foreach** *join tuple* $j \in \mathcal{J}$ *of* R **do**
3 $\quad \bar{p}$ = projected tuples associated with j tupleGeneratorForInsDel(j, $\bar{p}$)
4 **end**
5 **return** *v-diff*

Algorithm 6: Materialized View Generation

Chapter 7

Optimizations

We next present two optimizations that can be applied to GraphIVM to further increase its performance. The first optimization, referred to as *look-ahead* accelerates the view diff generation and view materialization process by adding to the join graph information on which partial paths can be extended to complete paths. The second optimization consists in replacing the general algorithms presented in Chapter 6 with code specifically crafted for a particular view; a process referred to as *view-specific compilation*.

7.1 Look-Ahead

Both the view diff generation and view materialization algorithms rely on finding all complete paths that contain a particular join tuple j. Currently both algorithm find these complete paths by iteratively exploring all partial

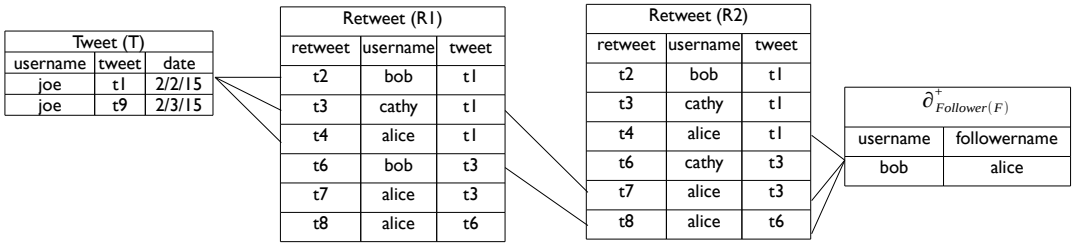


Figure 7.1: Example of how GraphIVM uses look-ahead to reduce the size of intermediate results

paths containing j and trying to extend them to a complete path. To accelerate this process, we introduce the concept of the *look-ahead*; a flag attached to a join tuple, signifying whether the join tuple is part of a complete path.

We differentiate between two variants of the look-ahead; the local look-ahead (LLA) and the global look-ahead (GLA). Given a join tuple j, its local look-ahead specifies whether j shares each of its vertex values with other join tuples. Thus the local look-ahead allows the algorithm to infer whether a partial join path containing only this join tuple can be extended in all directions by one join tuple. On the other hand the global look-ahead of a join tuple j specifies whether j is member of a complete path.

The look-ahead information can be used to quickly prune partial paths that cannot be extended to complete paths. We next illustrate this through an example:

Example 14 ***Look-Ahead.*** *Consider the following view TweetReach, which tracks the users to which a second degree tweet can reach:*

```
CREATE VIEW TweetReach as
SELECT T.username, T.tweet, F.user
FROM Tweet T, Retweet R1, Retweet R2, Follower F
WHERE T.tweet = R1.tweet
AND R1.tweet = R2.tweet
AND R2.username = F.username
```

Consider a base table diff $\delta_F^+(\langle bob, charlie\rangle)$ *for relation Follower. To create the view diffs corresponding to this base table diff, the view diff generation algorithm will start a BFS from* δ_F^+*, trying to find all complete paths that contain it. Figure 7.1 graphically depicts this search. As a first step it considers tuple* $r_1 = R2(\langle t4, charlie, t1\rangle)$*. Without look-ahead information, the view diff generation algorithm will have to include* r_1 *in the intermediate join result. However, querying the join tuple* $jt(r_1, J_{R2,F}) = \{\langle t4\rangle, \langle t1\rangle\}$ *for its LLA will reveal that it is not further connected and the view diff generation algorithm will safely ignore* r_1 *thereby reducing the intermediate join result size. Let us now consider the tuple* $r_2 = R2(\langle t8, charlie, t6\rangle)$*. Its local look-ahead is not very informative, since* r_2 *is connected to* $r_3 = R1(\langle t7, charlie, T3\rangle)$*. Thus in the presence of only LLA the algorithm would have to consider* r_2 *only to disqualify it when it reaches* r_3 *(which is not connected further). However, in the presence of GLA the fact that* r_2 *cannot lead to a complete path can be deduced simply by querying the GLA of the join tuple* $jt'(r_2, J_{R2,F}) = \{\langle t8\rangle, \langle t6\rangle\}$*. Since* jt' *is not a part of any complete path, therefore* r_2 *can be ignored as well.*

Maintenance of local and global look-ahead. The astute reader

may be wondering how these look-aheads are maintained when the join graph is updated. Since the LLA is based on knowledge only of the neighbors of a join tuple, it can be maintained eagerly by the join graph maintenance algorithm while processing a base table diff.

On the other hand, since the GLA of a join tuple j is based on a more global knowledge of the existence of complete paths containing j, its maintenance would require traversing the graph. To avoid paying the cost of this traversal every time the join graph is updated, GraphIVM allows GLA information to be partially invalidated after changes to the join graph. The GLA of a join tuple will again be made consistent whenever the particular tuple is visited by the graph traversal algorithm (either as part of the the view diff generation algorithm or as part of the view materialization algorithm). This allows the lazy maintenance of GLA information with negligible cost.

GraphIVM is executed by default with GLA enabled, as our experimental results presented in Section 9.6 have shown that GLA can have a profound positive impact in the performance of the system.

7.2 View-specific Compilation

There are in general two ways of coding an IVM approach. The first approach consists in writing a system containing generic code that can maintain any view. The second approach consists in writing a system which takes the view

as input and generates view-specific maintenance code. The second approach, which we call *view-specific compilation* has been shown to perform significantly better than the non-compiled approach in prior work [25, 3].

Based on these observations, we built in addition to the generic GraphIVM system (referred from now on as the default or non-compiled approach), also a compiled version of the former. Generating view-specific maintenance code allows for some natural optimizations that come from knowledge of the view. In particular, the compiled version of GraphIVM benefited among others from the following two optimizations: First, knowing the view and its schema, allows for the generation of code that uses primitive data types instead of generic objects. This not only improves memory efficiency, but may also enable additional compiler optimizations. Second, knowing the view implies knowing the maximum path length in a join graph. This information can be leveraged to code the BFS join graph traversal through a set of nested loops that are more amenable to compiler optimizations than general queue-based BFS approaches.

Our experimental results discussed in Chapter 9 indicate that producing view-specific code leads to both significantly better performance and lower memory requirements than generic non-compiled approaches.

Chapter 8

Alternative IVM Approaches

We next introduce the state of the art in IVM to which we will be comparing GraphIVM against. The first alternative, is traditional IVM without any auxiliary caches. This corresponds to the state of the art in IVM before approaches with aggressive materialization of auxiliary views where introduced. The second alternative is the DBToaster system [3]. Employing several optimizations, including among others aggressive materialization and view-specific compilation it has been shown to outperform existing IVM approaches. However, similarly to all other IVM systems with auxiliary caches we are aware of, it employs relational auxiliary views, in contrast to GraphIVM's non-relational join graph. Finally, we introduce a third alternative, which is inspired from the join graph but uses a relational data structure instead. This alternative, referred to as the *Full Outer Join Table* approach captures the same information

maintained by the join graph in the form of a full outer join table.

We next briefly describe each of the three approaches. The experimental evaluation of GraphIVM against each of these systems is presented in Chapter 9.

8.1 Classic IVM

This corresponds to traditional IVM approaches without the use of any auxiliary views. A typical IVM system has the following architecture: Given a set of base table diffs, encoded as tuples of a *base diff table* Δ, they compute a diff query Q that operates on both the diff table Δ and the base tables and computes the *diff table* for the view.

However, by not materializing auxiliary views, many invocations of the same diff query may be recomputing over and over the same partial results. GraphIVM avoids such recomputations by keeping materialized intermediate results in the form of the join graph. We next illustrate this through an example.

Example 15 *Let us apply the classic IVM technique to the view RetweetTracker of our running example. Ignoring projections for now the view definition can be written as* $User(U) \bowtie_{U.userId=T.userId} Tweet(T) \bowtie_{T.tweetId=R.retweetTweetId} Retweet(R)$. *If we consider deltas on Retweet, the diff query executed by classical IVM will be* $User(U) \bowtie_{U.userId=T.userId} Tweet(T) \bowtie_{T.tweetId=R.retweetTweetId} \Delta^m_{Retweet}$ *where* m *is the type of the modification. For every diff table instance* $\Delta^m_{Retweet}$ *classic IVM*

would need to repeatedly recompute the join $User(U) \bowtie_{U.userId=T.userId} Tweet(T)$ *(or at least the part of it relevant to the diff).*

8.2 DBToaster

DBToaster [3] represents the state of the art IVM system employing a variety of optimizations that make it outperform prior IVM approaches. Of particular interest for this work are two optimizations:

The first is the auxiliary view materialization strategy used by the system. DBToaster creates an entire lattice of auxiliary views such that one view can be efficiently maintained from other views in this lattice. However, all these auxiliary views are relational, leading to redundancy that not only negatively affects the memory requirements of the system but also the maintenance time, as one change has to be reflected in many views. We next illustrate this through an example:

Example 16 *Ignoring projections for now, the view RetweetTracker of our running example can be written as* $User(U) \bowtie_{U.userId=T.userId} Tweet(T) \bowtie_{T.tweetId=R.retweetTweetId} Retweet(R)$. *To maintain this view, DBToaster considers creating the following auxiliary views* $V_1 = User(U) \bowtie_{U.userId=T.userId} Tweet(T)$, $V_2 = Tweet(T) \bowtie_{T.tweetId=R.retweetTweetId} Retweet(R)$ *in addition to the base tables. It can be easily seen that a diff* $\delta^m_{Retweet}$ *on Retweet will have to be joined both with* V_1 *to maintain RetweetTracker and*

with the base table Tweet to maintain V_2. Thus the redundant representation of tuples employed by DBToaster leads in general to more maintenance.

A second important optimization is the generation of view-specific maintenance code. The inclusion of this optimization in GraphIVM not only improves the performance of the system, but as we will see next, also makes it easier to discern the difference in performance achieved by maintaining a non-relational auxiliary view instead of the relational views employed by DBToaster.

8.3 Full Outer Join Table

By looking at the join graph, one can easily see that what it captures are all partial join paths of the underlying database (up to tuples that do not satisfy the selections). Although the join graph represents such information in non-relational form, there is a relation that captures the same information; this is the full outer join equivalent of the original query (i.e., the original query where all inner joins have been replaced by full outer joins).

To check how an approach that captures the same information as the join graph but in relational form would perform, we introduce the Full Outer Join IVM system: a system, which keeps the full outer join equivalent of the view. It should be obvious that even in this case, the relational structure of this auxiliary view introduces significant redundancy, which leads to both space and time inefficiencies. Time inefficiencies are primarily caused by the need to

Timeline		
username	followername	tweet
joe	bob	t1
joe	alice	t1
joe	cathy	t1
joe	bob	t1
joe	alice	t1
joe	cathy	t1
cathy	bob	-
bob	alice	-
cathy	alice	-

Figure 8.1: Full outer join representation of the view Timeline

de-duplicate base table tuples to process a base table diff. Let us see this with an example.

Example 17 *Let us consider the following view Timeline and the database instance shown in Figure 5.2a.*

```
CREATE VIEW Timeline AS
SELECT username, followername, tweet
FROM Follower F, Tweet T
WHERE F.username = T.username
```

A full outer join implementation of this view is shown in Figure 8.1. If a tuple $\delta_F^+ = \langle joe, t10, 2/4/15 \rangle$ *is added to* $Tweet$*, we do not know the set of tuples of followers the base table diff* δ_F^+ *should join with. This is because the full outer join table of Timeline contains 6 rows for user* $\langle joe \rangle$*, 3 of them repeated twice. We would need to de-duplicate these 6 tuples to get the unique set of 3* $Follower$ *tuples.*

Chapter 9

Experimental Evaluation

To see how GraphIVM performs in practice, we implemented it and experimentally measured its performance. Our experiments explore among others the performance of GraphIVM compared to other IVM approaches in both the publish-subscribe and data warehousing scenarios. They also explore the effect of the optimizations described in Chapter 7 on GraphIVM and the memory requirements of the system.

9.1 Compared Systems

We compare GraphIVM against the three IVM approaches, described in Chapter 8: (a) the classic IVM approach without any auxiliary caches, (b) a full outer join table approach, and (c) DBToaster [3], which is a state-of-the-art IVM approach making among other extensive use of relational auxiliary

views. All systems compared are in-memory implementations apart from the classic IVM implementations, where for completeness we compared three different implementation of the classic IVM approach: The first is an in-memory implementation of the IVM problem that we coded independently of any DBMS. The second is a classic IVM approach built on top of an in-memory DBMS and the third is a classic IVM approach running on top of a disk-based DBMS. We next present all the systems used in our experimental evaluation. For each system we describe the runtime configuration used to run the corresponding system.

GraphIVM (GIVM/GIVMCPP). This corresponds to the implementation of the GraphIVM system. Unless otherwise stated, GraphIVM was executed with enabled look-ahead optimization. We distinguish between two versions of GraphIVM: The first version, referred to as GIVM is a general non-compiled implementation of GraphIVM written in Java. The second version, referred to as GIVMCPP is an implementation of GraphIVM that produces view-specific code as explained in Section 7.2. To enable an "apples-to-apples" comparison to DBToaster, which is implemented in C++, the compiled version GIVMCPP consists of view-specific C++ code.

The non-compiled Java version GIVM was run with JRE7 using 15GB heap with concurrent mark and sweep garbage collector on warm JIT. On the other hand, the C code produced from GIVMCPP was compiled using gcc4.8.2

with the -O3 flag (which are the same compilation options used for DBToaster as we will explain below).

Full Outer Join Table (OJT). This is an in-memory implementation of the full outer join approach, described in Section 8.3. Similarly to GIVM (and DIVM, discussed next) the full outer join approach was implemented in Java.

Classic IVM (DIVM). This is an in-memory implementation of classic IVM without auxiliary caches. DIVM contains indices on the base tables on the join attributes and computes view diffs using index-based tuple accesses. DIVM was written in Java and executed using the exact same configuration parameters as those used for GIVM.

Classic IVM using in-memory DBMS (H2). This is an implementation of classic IVM on top of the H2 in-memory database management systems (DBMS). In this case, we executed the diff query plan generated by classic IVM in H2. H2 was executed in embedded, in-memory mode with the following configuration options: (a) Auto analyze was disabled in order to reduce the latency of updates, (b) the query cache was disabled in order to avoid caching of the results between different runs of the same experiment, (c) temporary tables were used to reduce the overhead of logging, and (d) indices on the join attributes were built to enable the use of index-based joins.

Classic IVM using disk-based DBMS (Postgres). This is an imple-

mentation of classic IVM on top of the PostgreSQL disk-based DBMS. Although PostgreSQL is disk-based, we tried to move processing to memory by setting the buffer cache size to 2GB and the effective cache cache to 4GB (although we did not see PostgreSQL use more than 1GB of main memory at any time), and by using temporary tables. In addition, we turned off the auto analyze function in the interest of increasing performance, as we analyzed the relations manually before running the experiments. Moreover, we drove the PostgreSQL experiments from Java code, which was communicating to PostgreSQL using JDBC's prepared statements as we saw that this gave better performance than using PL/SQL. Finally, as in the case of H2, we specified indices on all join attributes to enable index-based join plans.

DBToaster (DBT). This is the official DBToaster implementation, which generates C++ code. The generated code was compiled using gcc4.8.2 with the -O3 flag, as recommended by the documentation on the system's website. All base tables in DBToaster were maintained as streams (i.e., specified as tables that may accept diffs) and diffs were injected to the system using trigger functions, which are more efficient than the alternative event-based mechanism also supported by the system.

Apart from the GraphIVM system that can natively support both the data warehousing and publish-subscribe IVM scenarios, all other systems were

executed in two different versions: the data warehousing version, which maintains the materialized view and is the default behavior of all such systems and the publish-subscribe version, which does not maintain the view but creates view diffs instead. To create the publish-subscribe version of DBToaster we edited the compiled code to remove the code of maintaining the view and replaced it by a call to a dummy function that accepts as parameter the view diff and has an empty body (essentially simulating a client waiting to consume the output diffs).

All experiments were performed on a PC with an Intel i7 3.2Ghz CPU and 16GB of RAM running Ubuntu 14.04. All the experiments were run on warm caches.

9.2 Datasets

To compare the systems on a social networking dataset, created using the BSMA-Gen [38] data generator, which was developed for the Benchmarking in Social Media Analytics (BSMA) benchmark [36]. BSMA-Gen produces a set of tweets and retweets based on a network for followers. To model a real social network, the tweet generation follows a Poisson process, while the retweet generation follows a power law in combination with a time decay function [38]. Using the output of the generator, we created a database instance, following the schema shown in Figure 9.1.

To check the scalability of the systems for simple queries, while being able to run the systems without running out of memory for more complex queries, we used BSMA-Gen to generate two datasets of different sizes. The first dataset, referred to as the *large* dataset was created by calling the BSMA-Gen for 1M Users. This resulted in 6M tweets and 7M retweets of 1M users with 110M followers. To make sure that this dataset fits in memory, we subsequently pruned the follower table down to 11M tuples in a random fashion. The resulting table sizes are summarized in Figure 9.2a. As we will see, this dataset was used to run relatively simple queries. To test how the systems scale for more complex queries (with large fanout and/or a large number of joins), we scaled this dataset down by a factor of 100. The table sizes of the resulting dataset (referred to as the *small* dataset) is shown in Figure 9.2b.

9.3 Queries

Given the social networking schema of Figure 9.1, we designed a set of nine queries over these schema to be maintained. Table 9.1 and 9.2 shows for each query the corresponding relational algebra description and a textual description explaining its semantics. The first three queries (queries Q1-Q3) correspond to simple Select-Project-Join (SPJ) queries that provide useful information in a social networking setting. The next three queries (queries Q1Agg-Q3Agg) are the aggregated versions of the previous queries. Finally the last

```
User(userid)
Follower(userId, followerId)
Tweet(userId, tweetId, tweetDate)
Retweet(userId, tweetId, tweetDate, retweetTweetId)
```

Figure 9.1: Schema of BSMA datasets

Relation	Tuples
Users	1M
Follower	11M
Tweet	6M
Retweet	7M

(a) Large BSMA dataset

Relation	Tuples
Users	10K
Follower	1M
Tweet	60K
Retweet	70K

(b) Small BSMA dataset

Figure 9.2: Size of each relation (in tuples) for the two BSMA datasets used in the experiments

three queries shown on the table are synthetic queries used to check the performance of the approaches under varying parameters.

9.4 IVM of Simple Queries

Our first experiment explores the performance of the different IVM approaches in maintaining simple queries (i.e., queries that have in this case at most two joins). To this end, we used all systems presented in section 9.1 apart from OJT[1] to maintain the three simple SPJ queries Q1-Q3 and their three

[1]The full outer join table approach (OJT) was considered in the fanout experiments described in Section 9.5.1 and found to perform worse than even classic IVM without auxiliary views (DIVM). For this reason it was not further considered in the experiments.

Table 9.1: Queries used in the experiments

Query ID	**Relational Algebra Expression**	**Description**
Q1	$User(U) \bowtie_{U.userId=T.userId} Tweet(T)$	Tweets associated with each user
Q2	$Follower(F) \bowtie_{F.userId=T.userId} Tweet(T)$	Tweets shown to a user on her timeline
Q3	$User(U) \bowtie_{U.userId=T.userId} Tweet(T)$ $\bowtie_{T.tweetId=R.retweetTweetId} Retweet(R)$	Retweets of tweets created by a user
Q1Agg	$\gamma_{U.userId;COUNT(T.tweetId)}$ $(User(U) \bowtie_{U.userId=T.userId} Tweet(T))$	Number of tweets per user
Q2Agg	$\gamma_{F.FollowerId;COUNT(T.tweetId)}$ $(Follower(F) \bowtie_{F.userId=T.userId} Tweet(T))$	Number of tweets shown to a user on her timeline
Q3Agg	$\gamma_{U.userId;COUNT(R.retweetId}$ $(User(U) \bowtie_{U.userId=T.userId} Tweet(T)$ $\bowtie_{T.tweetId=R.retweetTweetId} Retweet(R))$	Number of retweets, tweets of a user generated

Table 9.2: Queries used to measure the effect of fanout and join chain length

Query ID	**Relational Algebra Expression**	**Description**
QFanout	$Follower(F_1) \bowtie_{F_1.userId=F_2.followerId} Follower(F_2)$ $\bowtie_{F_2.userId=T.userId} Tweet(T)$	Tweets seen by second degree followers
QJoinChainLengthFollower	$Follower(F_1)(\bowtie_{F_{i-1}.userId=F_i.followerId} Follower(F_i))_n$ $\bowtie_{F_n.followerId=T.userId} Tweet$	Tweets seen by $n+1$ degree follower
QJoinChainLengthRetweet	$Retweet(R_1)(\bowtie_{R_{i-1}.tweetId=R_i.retweetTweetId} Retweet(R_i))_n$ $\bowtie_{R_n.userId=F.userId} Follower(F)$	Users which saw $n+1$ degree Retweets of a tweet

aggregated equivalents Q1Agg-A3Agg. To mimic the workload of a social networking system, we used the large dataset and considered the last 50,000 tuples added by the BSMA-Gen to tables Tweet and Retweet as diffs that were given as input to each of the systems [2].

Figures 9.3a and 9.3b show the time required by each system to maintain the 50,000 diffs for the publish-subscribe and data warehousing scenario, respectively. Missing bars indicate that the corresponding system ran out of memory during the experiment. By looking at the results, we can make the following observations:

GraphIVM vs Other Systems. We observe that the GraphIVM system always outperforms classic IVM approaches that do not employ any auxiliary caches. The speedup factor depends on the implementation of the specific approach. If the classic IVM approach is built on top of the disk-based PostgreSQL, then the speedup of GraphIVM is 10-100, if it is built on top of the in-memory H2, the speedup is 5-10 and if it is a specialized in-memory implementation (DIVM), the speedup is 2-3.

The comparison of GraphIVM to DBToaster is more interesting (in this case we compare the compiled C++ version GIVMCPP of GraphIVM, as DBToaster also produces C++ code). On SPJ queries (queries Q1-Q3) GraphIVM always outperforms DBToaster. For aggregate queries the picture is

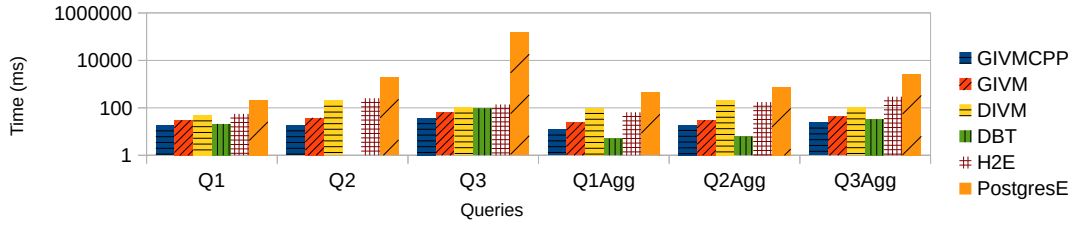
[2]BSMA-Gen creates timestamps for tweets and retweets, allowing us to infer which were the latest tweets and retweets that were added to the system.

mixed: DBToaster outperforms GraphIVM for queries Q1Agg and Q2Agg, but falls behing the latter for query Q3Agg. This behavior is due to the extremely simple nature of Q1Agg and Q2Agg. Both queries consist of a single join and thus DBToaster does not have to create any auxiliary view, which corresponds to the best case for the system. As we can see in the case of Q3Agg, which has two joins and as will become even more obvious in the following Sections, when the queries involve longer join chains, GraphIVM easily outperforms DBToaster.

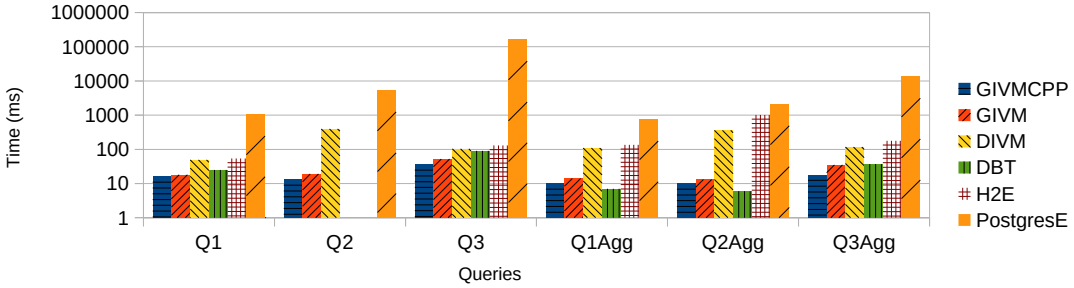
Publish-Subscribe vs Data Warehousing Scenario. In general systems have to carry out more work in the data warehousing scenario as compared to the publish-subscribe scenario, as they have to not only compute the diffs (which they do anyway during the IVM process) but also apply them to the view to maintain it. Interestingly, GraphIVM breaks this pattern by performing better in the data warehousing scenario. The reason for this discrepancy is the unique architecture of GraphIVM: Since GraphIVM does not maintain the view as a separate data structure but as part of the join graph, in the data warehousing scenario it does not have to perform any additional view maintenance. On the other hand though, since it does not internally generate view diffs, it has to incur the cost of generating them from the join graph in the case of the publish-subscribe scenario. This leads to the cost of the publish-subscribe scenario exceeding that of the data warehousing scenario in the case of GraphIVM. Finally, even though GraphIVM does not have to pay the cost of

maintaining a relational view on updates, it has to incur the cost of generating a relational version of the view from the join graph in the data warehousing scenario, whenever the clients want to access the view. We explore this cost of view materialization in Section 9.8.

Accounting for Set-oriented Processing in DBMSs. Our experiments so far simulated eager view maintenance, in which each IVM system is invoked for every single incoming base table diff. However, it is well known that DBMSs can benefit from processing a set of tuples at a time. To see whether this makes a difference in the relative performance of the systems, we ran a variant of the above experiment in which all 50,000 input diffs were given to each system as input at once. This simulates a lazy IVM scenario. The resulting IVM times are shown in Figures 9.4b and 9.4a for the data warehousing and publish-subscribe scenario, respectively. The times of GraphIVM, DIVM and DBToaster remain the same as before, as they internally operate at one tuple at a time. The times of IVM approaches built on top of DBMSs generally improve with the biggest improvement witnessed by PostgreSQL, which improves by an order of magnitude over the eager maintenance. However, even with this improvement GraphIVM still outperforms these systems.



(a) Publish-subscribe scenario



(b) Data warehousing scenario

Figure 9.3: IVM of simple queries (eager maintenance)

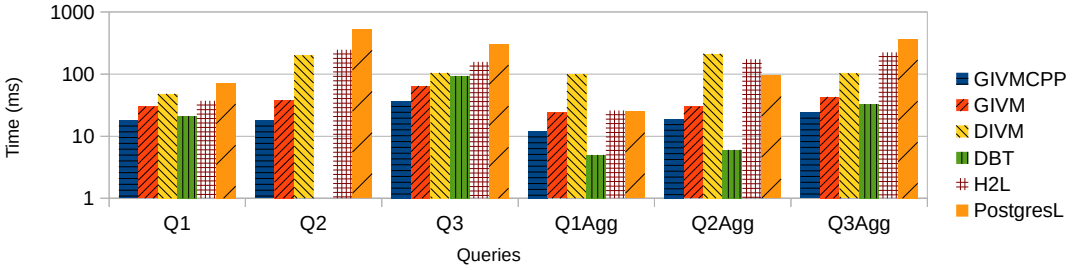
9.5 Effect of varying parameters

We next explore the effect of varying two parameters, that are commonly characterizing the structure of a join view: the fanout and the number of joins.

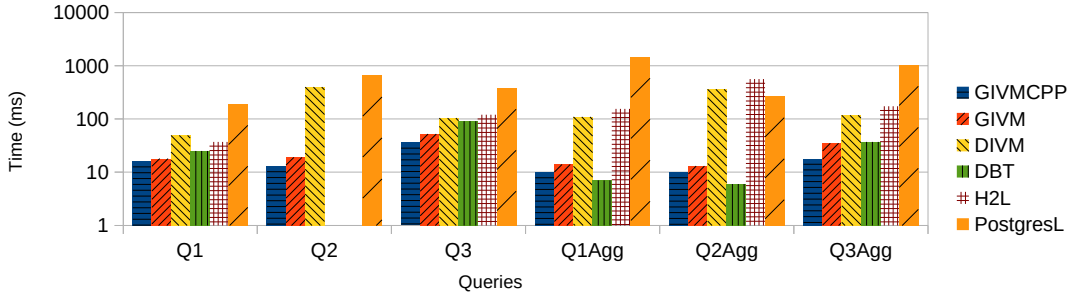
9.5.1 Varying Fanout

Given an $SPJA_p$ query Q and an input diff δ, we define the fanout of Q w.r.t. δ to be the number of view diffs that should be generated as the result of maintaining the single base table diff δ.

To control the effect of this parameter, we executed the following experiment. Starting from the small BSMA dataset, we created 50,000 new users and

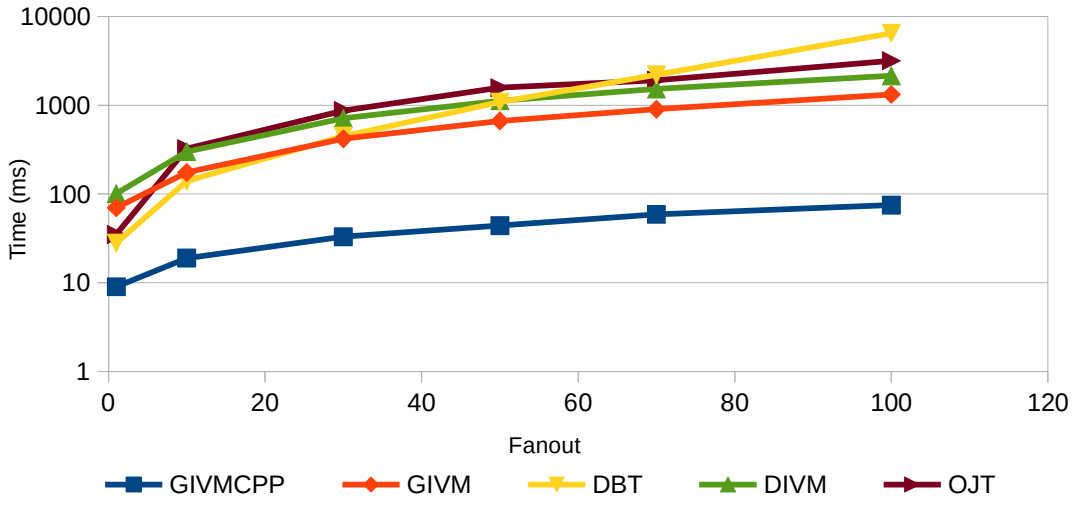


(a) Publish-subscribe scenario

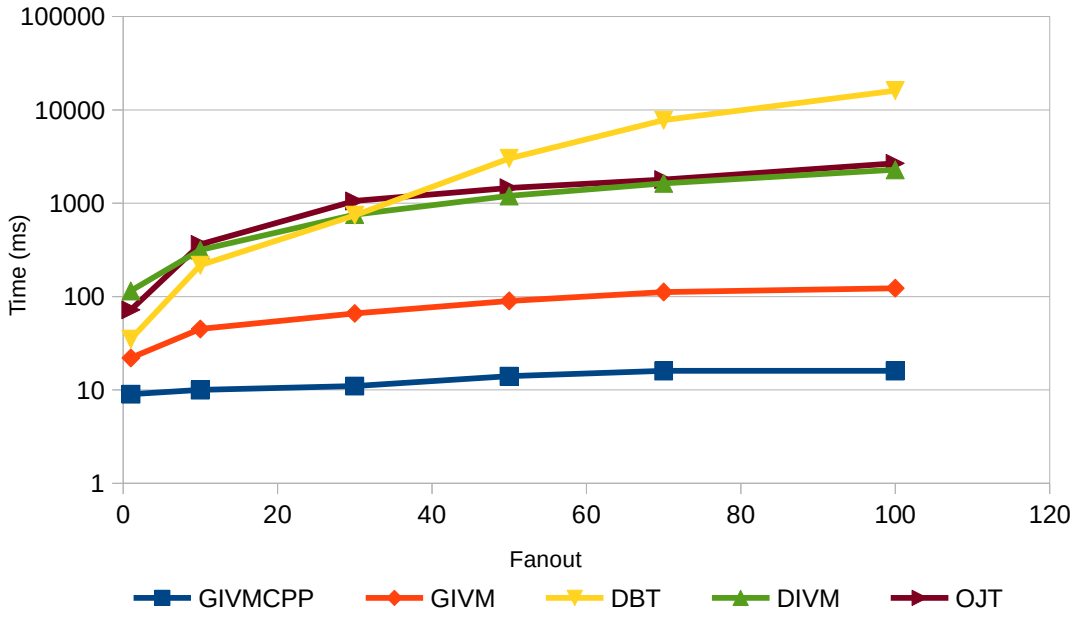


(b) Data warehousing scenario

Figure 9.4: IVM of simple queries (lazy maintenance)



(a) Publish-subscribe scenario



(b) Data warehousing scenario

Figure 9.5: IVM performance for varying fanout

for each such user u we created n Follower tuples (i.e., we created n tuples of the form Follower(u, f'), where f' is a new value), essentially specifying that each such user has n followers. We then created 50,000 new insert diffs for the Tweet relation, each inserting one tweet for each newly created user u and we asked the systems to maintain the view QFanout as given in Table 9.2. It is easy to see that each input diff generates n view diffs, essentially making the parameter n the fanout of the view w.r.t. the input diff.

Figures 9.5a and 9.5b show the time required by the GraphIVM (both non-compiled and compiled versions), the in-memory classic IVM, DBToaster and Full Outer Join approach to maintain the view for fanouts ranging from 1 to 100 for the publish-subscribe and data warehousing scenario, respectively.

For the publish-subscribe scenario (Figure 9.5a), all approaches require more time to perform IVM as the fanout increases. GraphIVM has been found to perform better than any other Java-based approaches. The full outer join approach has been found to perform worse than even classic IVM and therefore we did not consider it in the rest of the experiments. DBToaster is shown to perform better than GIVM for small values of the fanout, but keep in mind that this corresponds to the non-compiled Java-based version of GraphIVM. If we compare DBToaster to the compiled C++-based version of GraphIVM (denoted as GIVMCPP), we see that the latter outperforms DBToaster by a speedup that is most of the time an order of magnitude and increases with fanout.

The difference between GraphIVM and the other systems becomes even more pronounced in the data warehousing scenario (Figure 9.5b). While all approaches perform worse with increasing fanout, GraphIVM remains almost constant, as it does not need to create output diffs and is thus not affected by the fanout. The slight decrease in performance is probably due to engineering details (potentially decreased performance of the hash-based indices).

9.5.2 Varying Number of Joins

In this experiment, we investigate the relationship between the number of joins in the view definition and the time required to incrementally maintain the views. To explore this relationship we used the queries QJoinChainLength-Follower and QJoinChainLengthRetweet of Table 9.2 including a chain of n Follower and n Retweet joins, respectively. We varied n between 1 and 4, leading to a variation of the number of joins per query between 2 and 5. For query QJoinChainLengthFollower, we considered the insertion of 50,000 Tweet tuples for a particular user, while for query QJoinChainLengthRetweet we considered the insertion of 50,000 follower tuples for a particular user. Both experiments were run on the small BSMA dataset.

These two queries represent the two different effects an increase in the number of joins may have. Introducing more joins in a query may either increase the fanout (if introducing a join leads to the construction of additional tuples)

Table 9.3: Effect of number of joins on fanout for queries QJoinLengthFollower and QJoinLengthRetweer

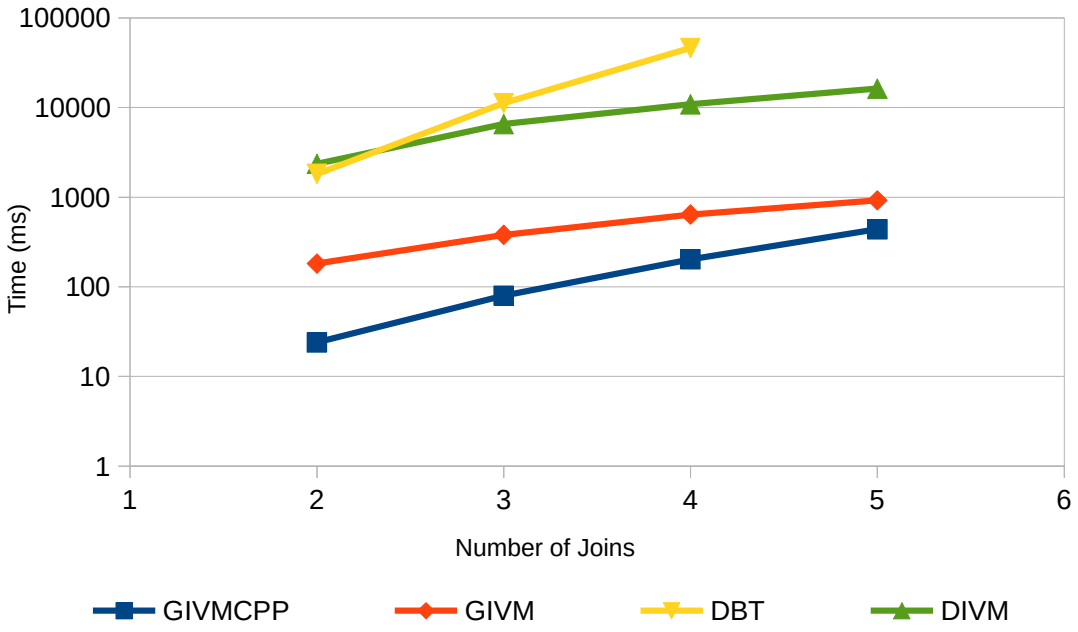
Number of joins	2	3	4	5
QJoinLengthFollower Fanout	182	381	445	545
QJoinLengthRetweet Fanout	110	56	24	11

or decrease it (if the join acts like a filter). In our case increasing the join length leads to an increase of fanout in the case of QJoinChainLengthFollower (because a follower has in general more than one followers) and in a decrease of fanout in the case of QJoinChainLengthRetweet (because a lot of retweets are not further retweeted).

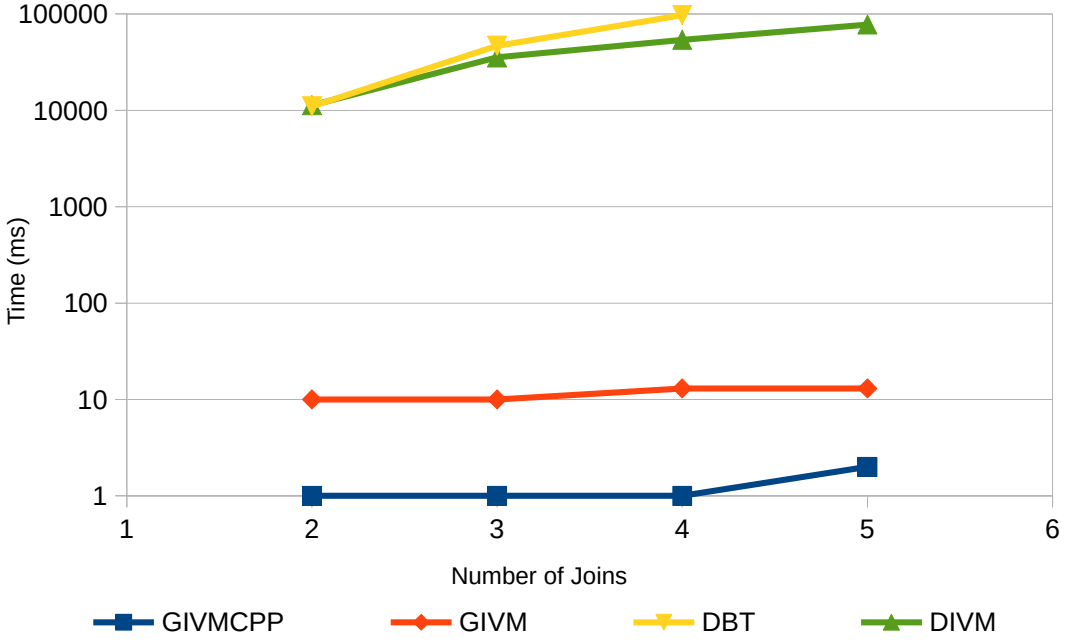
Table 9.3 shows the fanout for each query as the number of joins increases. We next discuss each of the two cases:

Increasing fanout with increasing number of joins (query QJoinChainLengthFollower). Figures 9.6a and 9.6b show the IVM times for this case for the publish-subscribe and data warehousing scenario, respectively. The increase in fanout makes all systems require more time for IVM as the number of joins increases. However, both versions of GraphIVM perform always better than all other systems.

Decreasing fanout with increasing number of joins (query QJoinChainLengthRetweet). Figures 9.7a and 9.7b show the IVM times for this case for the publish-subscribe and data warehousing scenario, respectively. The decrease in fanout makes GraphIVM become better with increased number of



(a) Publish-subscribe scenario



(b) Data warehousing scenario

Figure 9.6: IVM performance for varying number of joins when increasing join number leads to increased fanout (Query QJoinChainLengthFollower)

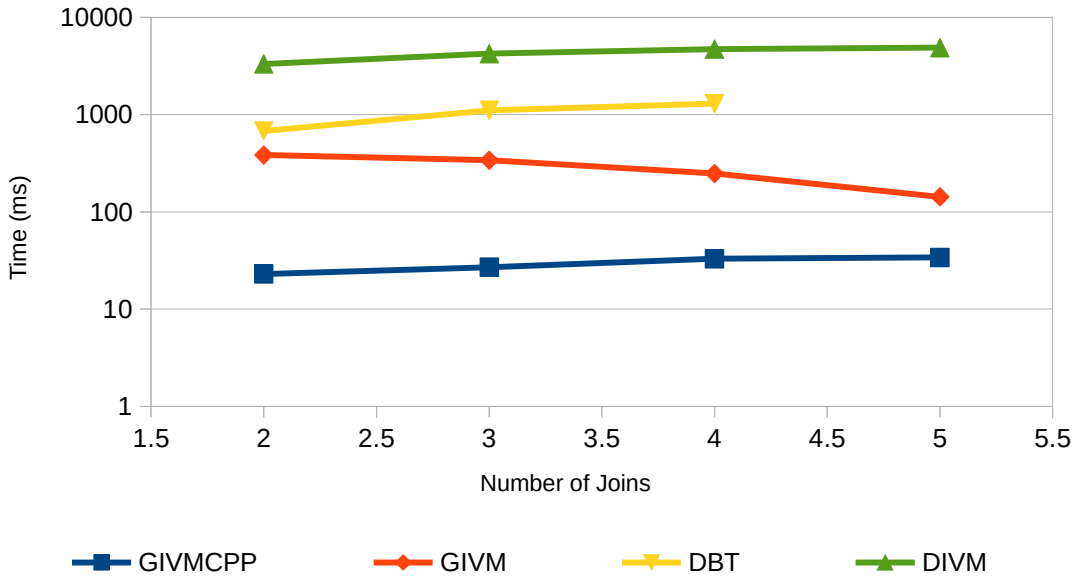
Table 9.4: Effect of number of joins on number of auxiliary views maintained by DBToaster for queries QJoinLengthFollower and QJoinLengthRetweer

Number of joins	2	3	4	5
DBToaster: # Views Created	5	7	9	SOE
DBToaster: # Views Updated	3	4	5	SOE

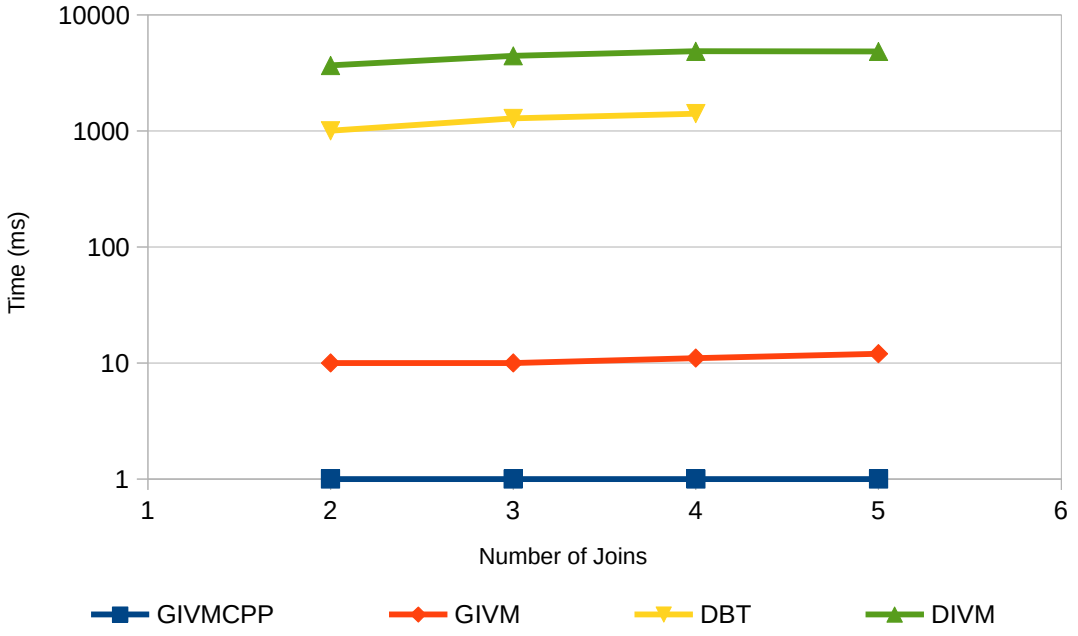
joins. Interestingly, DBToaster becomes worse, as the decreased fanout is offset by the need to maintain more auxiliary views as the number of joins increases. Table 9.4 shows the number of views created and updated by DBToaster in the maintenance of each of the queries for varying number of joins. No numbers are reported for 5 joins, as DBToaster crashed with a Stack Overflow Error (SOE). Helped by its compact, non-relational auxiliary view (and as we will see also by the look-ahead optimization), GraphIVM not only does not run out of memory but even improves its performance with increasing number of joins in this case.

9.6 Effect of Look-Ahead Optimization

To check how much the performance of GraphIVM is a result of the look-ahead optimization, we next ran the same two experiments of varying number of joins for GraphIVM with no look-ahead, with local look-ahead and with global look-ahead enabled. We denote the respective systems by GIVM (NLA), GIVM (LLA) and GIVM (GLA). Figures 9.8a and 9.8b show the IVM times of these systems for varying number of joins for queries QJoinChainLengthRetweet

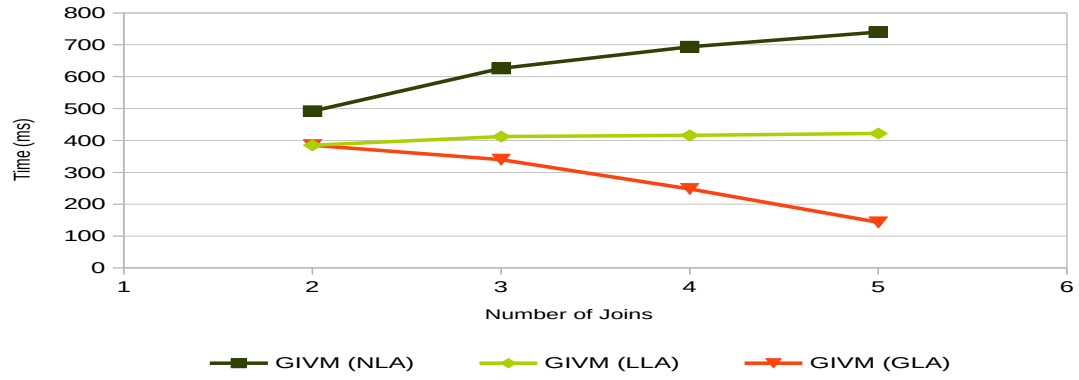


(a) Publish-subscribe scenario

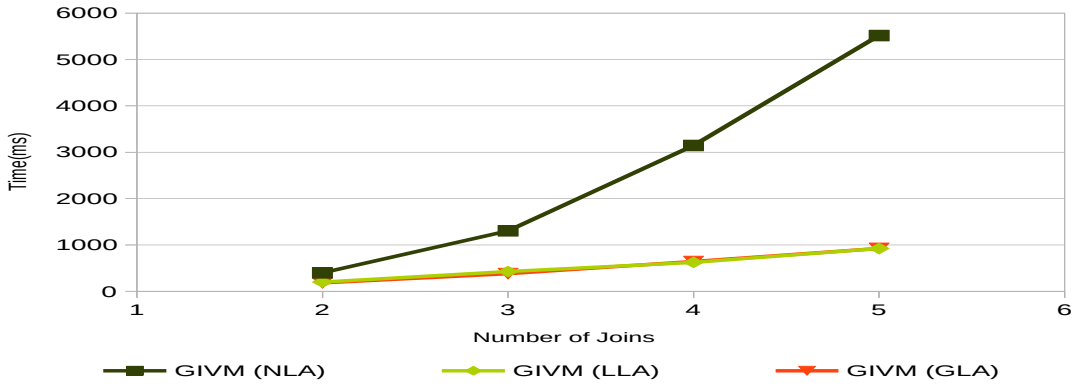


(b) Data warehousing scenario

Figure 9.7: IVM performance for varying number of joins when increasing join number leads to decreased fanout (Query QJoinChainLengthRetweet)



(a) QJoinChainLengthRetweet



(b) Query QJoinChainLengthFollower

Figure 9.8: Effect of look-ahead optimization for varying number of joins for queries QJoinChainLengthRetweet and QJoinChainLengthFollower (Publish-subscribe scenario)

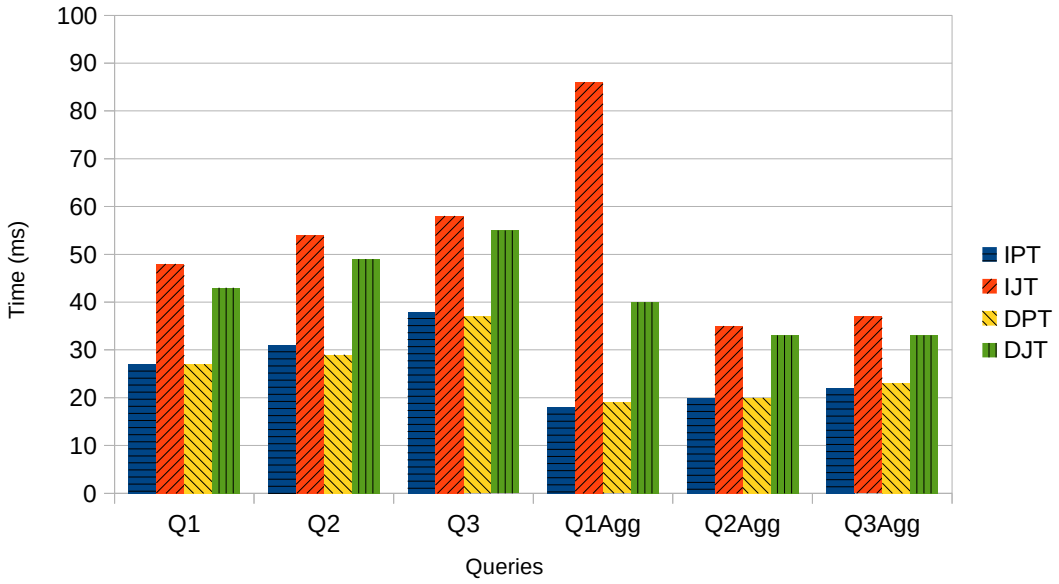
and QJoinChainLegthFollower, respectively. The times are shown only for the publish-subscribe scenario, as in the data warehousing scenario, GraphIVM does not create view diffs and is thus not affected by the presence or absence of the look-ahead optimization.

Let us first look at QJoinChainLengthRetweet (Figure 9.8a). As we explained in the previous section, GraphIVM performs better as the number of joins increase when the global look-ahead is enabled. However, the picture changes when GraphIVM employs only local look-ahead or no look-ahead, in which case it performs almost identically or worse, respectively with increasing number of joins. The global look-ahead in this case, helps GraphIVM to quickly disregard during the BFS procedure paths that are not complete.

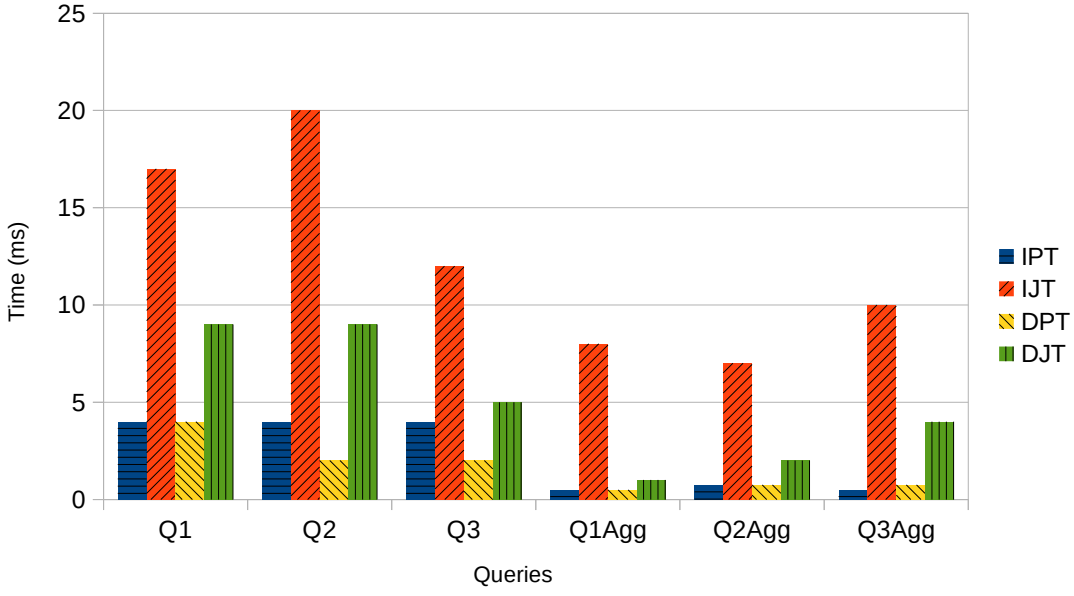
In the case of query QJoinChainLengthFollower (Figure 9.8b) GraphIVM requires always more time as the number of joins in the query increase. However, even in this case, global look-ahead proves to be extremely beneficial.

9.7 Relative Performance of Different Types of Modifications to Join Graph

As we discussed in the description of the join graph maintenance algorithm in Section 6.2, a base table diff may lead to different types of modifications to the join graph. In particular, it may lead to an insertion or deletion of a join



(a) GIVM



(b) GIVMCPP

Figure 9.9: Performance of different types of modification to the join graph of different queries

tuple or projected tuple. These join graph modifications incur in general different costs.

In this experiment, we compared these costs as follows: Starting from the large BSMA dataset and for the join graph of each query Q1-Q3 and Q1Agg-Q3Agg, we measured the time it takes to apply a certain modification time 50,000 times. The resulting times are shown for both the general non-compiled version of GraphIVM and for the compiled variant in Figures 9.9a and 9.9b, respectively. The labels IPT, IJT, DPT and DJT stand for insertion of projected tuple, insertion of join tuple, deletion of projected tuple and deletion of join tuple, respectively.

The experiment shows that the insertion of a join tuple takes more time than the insertion of projected tuples. Similarly for the deletion of respective tuples. The reason for this difference is that a modification of a join tuple on top of what is needed to modify a projected tuple also entails changes to the vertex values.

9.8 Performance of Materializing the View

As we discussed above, GraphIVM maintains the view internally as the join graph and thus does not have to pay the price of maintaining a separate relational view in the data warehousing scenario. On the other hand however, it will have to pay the cost of generating this relational representation of the

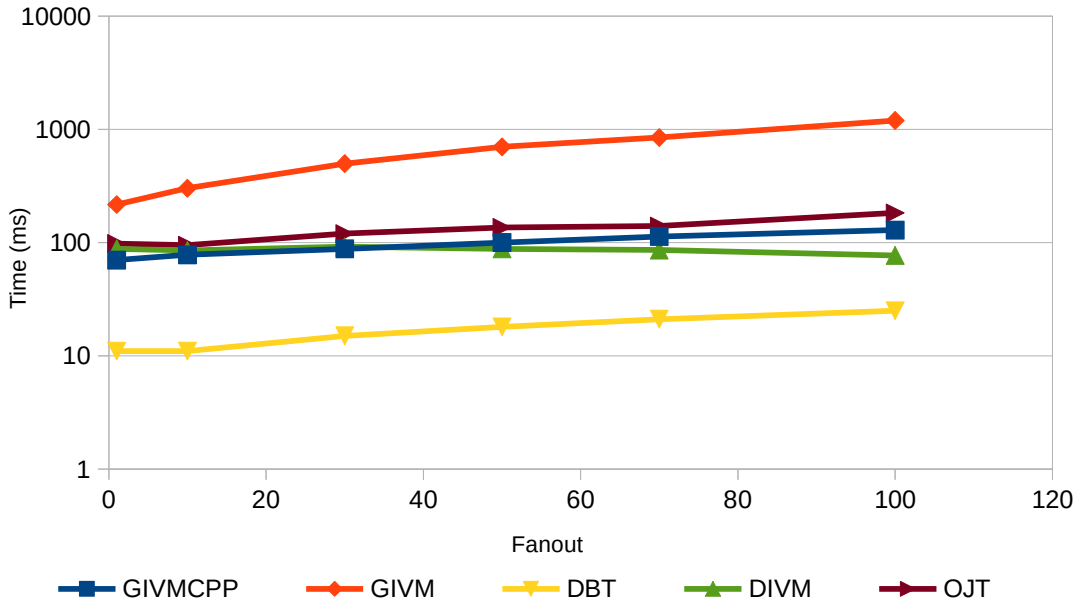


Figure 9.10: Time required to materialize (scan) the view that has resulted from the varying fanout experiment

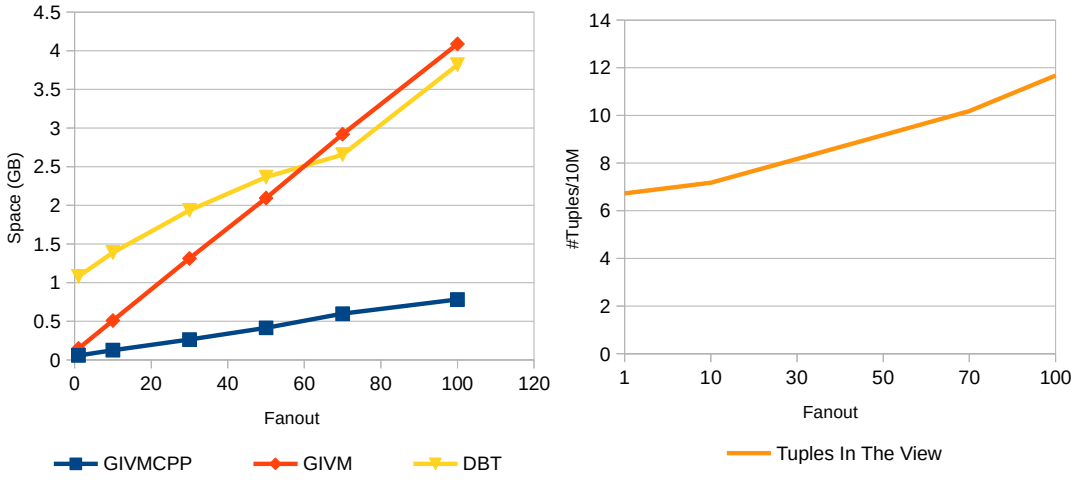
view from the internal join graph, when this view is accessed.

To check this view materialization cost, we ran an experiment in which the entire view was read from each of the systems. As for the view that was scanned, we used the final view after running the varying fanout experiment of Section 9.5.1. The resulting times are shown in Figure 9.10. Since GraphIVM has to reconstruct the relational view from the non-relational join graph by traversing the latter, it needs more time than other approaches, which store the view in a relational (or almost) relational format. However, keep in mind that on the other hand, due to this non-relational structure GraphIVM achieves much better IVM performance than other approaches.

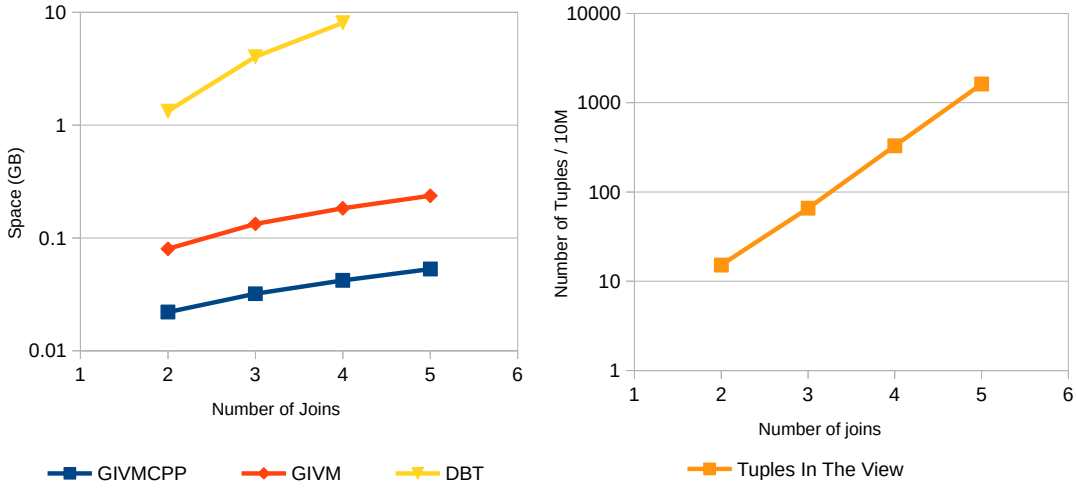
9.9 Memory Requirements

As a final experiment, we looked at the memory requirements of GraphIVMand compared it to the memory requirements of DBToaster. To check the memory requirements of both systems, we measured the memory used by each system at the end of the varying fanout experiment (described in Section 9.5.1) and the varying number of joins experiment on query QJoinChainLengthFollower (described in Section 9.5.2). The resulting space requirements for GIVM, GIVMCPP and DBT are shown in Figures 9.11a and 9.11b for the fanout and number of joins experiment, respectively. To see how the memory requirements of each system scales w.r.t. the number of tuples in the view, each figure also displays the size of the view in tuples.

We can see that the memory requirements of the compiled C++ implementation of GraphIVM are for both experiments much lower than the memory requirements of DBToaster. However, the most interesting observations come from the experiment varying the number of joins (shown in Figure 9.11b). While the memory requirements of DBToaster scale linearly with the number of tuples in the view, GraphIVM manages to scale a a sublinear pace. This is the result of the highly compressed join graph representation used by GraphIVM.



(a) Varying fanout (Query QFanout)



(b) Varying number of joins (Query QJoinChainLengthFollower)

Figure 9.11: Memory requirements of GraphIVM and DBToaster for varying fanout and number of joins

Chapter 10

Future Work

As part of our future work we plan to extend the presented work in two important directions. First, we would like to improve the scalability of GraphIVM by allowing it to store the join graph either partially on in its entirety on disk. Second, we would like to extend the approach to more expressive view definition languages. We discuss next each of these two extensions in detail.

10.1 Improving Scalability

GraphIVM is currently an in-memory system designed to demonstrate how non-relational caches can be used to accelerate IVM. While the in-memory implementation is sufficient for many use cases, it precludes the system from supporting IVM of very large datasets.

To address this issue, we are planning of extending the system to store

the join graph on disk and bring into main memory only the parts relevant to the processing being done. In such a system the various components of the join graph would be stored on disk and the existing indices or pointers would be changed to contain the block address and offset within the disk block where a component can be found.

However, in order for this scheme to work efficiently, we will have to address among others the following three major challenges: The first challenge is designing how data have to be laid out on disk to improve the I/O throughput. Data that are accessed together by the algorithm should ideally be stored in sequential parts of the disk. Fortunately, the algorithms discussed in Chapter 6 give some hints on expected access patterns. For instance, a join tuple is usually accessed together with the projected tuples attached to it, suggesting that both concepts should be co-located.

Another challenge is making sure that indices, which are predominantly used by the algorithms are as efficient as possible. This issue could be solved by making sure that the indices could reside in memory.

A third challenge is deciding exactly what to read into the main memory and cache. A lazy approach that transfers into the main memory only the tuples that are needed for a particular step of the algorithm may prove inefficient. Instead, the system could bring an entire part of the join graph that is expected to be accessed either during the particular invocation of the algorithm or in the

future. This suggests a hybrid approach where part of the join graph is accessed from the main memory while another part (that does not fit in memory) is accessed from disk on demand.

As part of our future work, we will be exploring whether using our approach-specific optimizations as well as other optimizations employed by DBMSs, it is possible to scale GraphIVM to larger datasets that do not fit in main memory, while maintaining its performance characteristics.

10.2 Extending the class of supported views

As discussed above, GraphIVM currently allows the efficient IVM of $SPJA_p$ views. To extend the applicability of our work to a larger number of views, we seek to extend the class of supported views. This extension could be done in different ways: First, we could extend the join operators to support not only equi-joins but also joins involving inequalities or arbitrary functions. Second, we could add support for selections that cannot be pushed all the way down to the base tables. This includes among others selections that act on the result of an aggregation. Both of these extensions could be handled by creating a view diff filter which acts on the results of view diff generator or view materialization algorithms (Algorithms 5 and 6, respectively).

Finally, we would also like to extend our framework to aggregations involving non-associative functions. To allow the efficient IVM of such views, we

could extend GraphIVM to allow the addition of user-defined data structures and corresponding user-defined functions designed to handle the IVM of aggregations involving such a non-associative function. For instance, the system could support $MAX(R.a)$ on attribute a of relation R on deletions, by storing the all values of a. This could be accomplished by creating a MAX-heap data structure and a corresponding user-defined function responsible for handling the IVM of aggregations involving the MAX function.

Chapter 11

Conclusion

We have seen how moving from relational to non-relational auxiliary views can lead to significant performance improvements in incremental view maintenance. The novel non-relational join-graph together with the look-ahead and view-specific compilation optimizations allow GraphIVM to significantly outperform state of the art IVM approaches in most common scenarios. In our future work we plan to extend the practical applicability of the system by improving its scalability and extending the supported view definition language.

I would like to thank Yannis Katsis and Yannis Papakonstantinou who co-authored this thesis with me.

Bibliography

[1] New tweets per second record, and how! `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`. Accessed: 2015-04-12.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[3] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[4] E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts. Everyone's an influencer: quantifying influence on twitter. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 65–74. ACM, 2011.

[5] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 61–71, 1986.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD 2000*, pages 379–390, 2000.

[7] A. Cheng, M. Evans, and H. Singh. Inside twitter: An in-depth look inside the twitter world. *Report of Sysomos, June, Toronto, Canada*, 2009.

[8] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[9] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/sql. *ACM SIGMOD Record*, 29(1):44–51, 2000.

[10] J. C. Freytag and N. Goodman. Translating aggregate queries into iterative programs. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 138–146, 1986.

[11] J. C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM Trans. Database Syst.*, 14(1):1–27, 1989.

[12] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Advances in Database Technology EDBT'96*, pages 140–144. Springer, 1996.

[13] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 1995.

[14] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD 1993*, pages 157–166, 1993.

[15] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT '97*, pages 98–112, 1997.

[16] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT '99*, pages 453–470, 1999.

[17] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[18] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[19] Y. Jin and R. E. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, DEBS 2003, Sunday, June 8th, 2003, San Diego, California, USA (in conjunction with SIGMOD/PODS)*, 2003.

[20] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing incremental view maintenance in nested data models. In *Database Programming Languages, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, August 18-20, 1997, Proceedings*, pages 202–221, 1997.

[21] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 613–624, 2010.

[22] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 95–104, 1995.

[23] L. Liu, C. Pu, and W. Tang. Correction to continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 12(5):861, 2000.

[24] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[25] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[26] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, pages 802–813, 2002.

[27] D. Quass. Maintenance expressions for views with aggregation. In *VIEWS*, pages 110–118, 1996.

[28] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 158–169. IEEE, 1996.

[29] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 23, 2006.

[30] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 447–458, 1996.

[31] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.

[32] O. Shmueli and A. Itai. Maintenance of views. In *ACM SIGMOD Record*, volume 14, pages 240–255. ACM, 1984.

[33] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 126–135, 1997.

[34] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[35] J. Widom. Research problems in data warehousing. In *CIKM '95*, pages 25–30, 1995.

[36] F. Xia, Y. Li, C. Yu, H. Ma, and W. Qian. Bsma: A benchmark for analytical queries over social media data. *PVLDB*, 7(13), 2014.

[37] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[38] C. Yu, F. Xia, Q. Zhang, H. Ma, W. Qian, M. Zhou, C. Jin, and A. Zhou. Bsma-gen: A parallel synthetic data generator for social media timeline structures. In *Database Systems for Advanced Applications*, pages 539–542. Springer, 2014.