UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Local-First Relation Views

Materialized views in SynQLite

Lars Marius Elvenes

INF-3981 Master's Thesis in Computer Science

“Blame the implementation, not the technique.”
–Tim Kadlec

Abstract

In today's digital landscape where cloud-oriented approaches are widespread and an integral part, local-first software emerges to offer an alternative [16]. It addresses concerns such as data control, privacy, offline capabilities, collaboration, and performance. The open-source relational database engine SQLite [28] is a fitting candidate for local-first software as it is not reliant on network connectivity. SynQLite [33] is an extension of SQLite that enables CRR (conflict-free replicated relations) support, applying CRDTs (conflict-free replicated data types) to allow replicating changes between sites without a dedicated coordinator. In this thesis, we implement materialized views [7] in SQLite on top of SynQLite. The views are incrementally maintained [15] to increase performance in comparison to complete refresh, and use data provenance [14] and causal length [34] in order to track changes across sites and determine how they should be applied to the views. Experiments comparing the incremental refresh against the complete refresh show that the incremental approach is generally faster, at the cost of extra storage usage.

In today's digital landscape, characterized by widespread cloud-oriented approaches, the concept of local-first software [16] has emerged as an alternative solution to address concerns related to data control, privacy, offline capabilities, collaboration, and performance. This thesis focuses on implementing materialized views [7] on top of SynQLite [33], an extension of the SQLite relational database engine, to enhance performance and support local-first software principles.

The SynQLite extension to SQLite offers CRR(conflict-free replicated relations) [34] support by applying CRDTs (conflict-free replicated data types) [19]. This allows for replicating changes between sites without a dedicated coordinator. By implementing incrementally maintained materialized views [15], we aim to increase performance compared to complete refresh approaches. Our implementation incorporates data provenance [14] and causal length [34] to track changes across sites and determine how they should be applied to the views.

Through experiments comparing the incremental refresh with the complete

refresh, we find that the incremental approach generally achieves faster performance, albeit with additional storage usage. These findings demonstrate the potential of materialized views within local-first software applications. This research contributes to the understanding and implementation of local-first software principles in the context of SQLite and SynQLite, paving the way for improved data management and collaboration in decentralized environments.

Acknowledgements

I would like to express my gratitude to my supervisor, Weihai Yu, for introducing me to the exciting topic of local-first views. Your guidance, expertise, and continuous support throughout the entire research process have been invaluable. Your insightful feedback and constructive criticism have greatly shaped the direction of this thesis and enhanced the quality of the work.

Special thanks go to my fellow classmates, who have created an engaging and enjoyable environment during this journey. Thank you for all the lunch breaks, discussions, and games of pool. I deeply appreciate our friendship and your lasting impact on my academic and personal development.

Contents

List of Figures

List of Tables

1

Introduction

In the field of relational databases, the concept of views [32] is a well-established and researched topic. These are virtual tables, storing a query definition and allowing its results to be accessed as if they were real tables. They are represented as seemingly normal relations, where data is ordered into rows and columns, but the results are calculated from scratch each time they are accessed.

In a distributed setting, the views may be derived from queries on data that resides in locations other than where the view is accessed from. Recalculating the view on each access leads to increased traffic due to fetching data from remote sites. A materialized view can accommodate this, by storing the results of the query as well [6] and allowing more efficient accesses. However, these types of views require some additional mechanisms for synchronizing and refreshing the data as changes to the data occur [18], and is also referred to as maintenance. One such type of refresh method is incremental maintenance [15], where only the altered data is refreshed, in contrast to a standard complete refresh, where everything is recalculated.

As different sites may perform different modifications to their data, one also needs some way of deciding which changes should win when conflicts occur. Conflict-free replicated data types (CRDTs) [19] have risen as a way of tackling these issues and ensuring that the results of changes in a distributed setting will be the same as for a non-distributed setting, in an eventual consistent matter. They have no centralized coordinator, and instead, each site will make use of history tables, clocks, and casual lengths to decide the combined results of

changes from multiple sites. CRRs (conflict-free replicated relations)[34] make use of CRDTs in relational database systems, and one concrete example of a CRR is SynQLite, an expansion on top of the widely used SQLite.

Local-first software [16] is a set of principles that aim to facilitate collaboration between users and retain user ownership at the same time. It attempts to take the best of both worlds of traditional offline systems and modern cloud services. Local-first software often makes use of technologies like CRDTs to store data and enable version control.

In this master's thesis, we look into handling local-first materialized views that are incrementally maintained and built upon SQLite and the extension SynQLite to support a distributed database system. Using SQLite for local-first software is a fitting choice, as it does not rely on internet connectivity, and SynQLite grants CRR [34] support, which applies CRDTs to relational databases. We make use of a new variant of provenance expressions to keep track of how the data of the base tables (points of origin) affects the view tables, which allows us to handle update operations in addition to inserts and deletes.

1.1 Problem statement

In this thesis, we will develop, on top of SynQLite, support for incremental maintenance of local-first views.

The overall goal is to have an implementation of materialized views in SynQLite. Users should be able to create views stored physically based on a given query on a specific site and refresh them on demand, pulling changes from a different given site. They should use an incremental maintenance approach that the user does not need to manage manually. The system should be able to handle concurrent changes to the same tables, and the view results should be the same as if the database was non-distributed.

A set of initial goals was later revised and reformulated into requirements. The most significant alterations are supporting UPDATE operations on the base tables, and using incremental maintenance. Updates operations on base tables should be reflected in the resulting view, and concurrent updates should also be handled with the results being the same as in a non-distributed setting.

The list of requirements is as follows:

- Views are materialized (results are stored physically)

- Views can be accessed without any additional calculations
- Views satisfy local-first properties and are persistent and accessible offline
- Views can be made using SELECT, PROJECT, and JOIN queries
- Supports insert, delete and update operations
- Can refresh from any available replicas
- Supports complete and incremental refresh
- Ensures eventual consistency using data provenance and causal length
- Users can manage views through Python or shell commands
- Refreshing does not require any manual resolution
- Refreshing is deferred

1.2 Scope

The scope of this project is to research and look into a solution for materialized views in SynQLite. It is not yet intended for production as there remains more work to be done to ensure a bug-free experience and a more full-fledged array of functionality.

The implementation of the views is limited to supporting the most fundamental operations in queries: select, project, and join (SPJ). It is able to refresh changes made from INSERT, DELETE, and UPDATE operations done on the base tables. The thesis does not look into making views querying other views as this requires generating deltas from the views themselves. It also does not support all SQLite functionality, such as outer joins [29], aggregations [24], etc. The limitation to SPJ queries is sufficient to display some of the capabilities of the system and how the different views can be handled. Sub-queries are another aspect this project does not look into, as it is out of scope.

The current implementation does not look into real network traffic, but instead simulates it using different database files located in different directories on a local machine. This should be sufficient in order to test and look into the ideas discussed in this project. It can later be expanded upon as SynQLite offers support for real network traffic, but for this thesis, it is defined as out of

scope.

1.3 Contributions

The project researches and implements incrementally maintained materialized views in SynQLite, which previously had no support for any type of materialized view. SQLite also has no internal support for materialized views in any way, meaning it has to be built from scratch. This research looks into and suggests a technique for dealing with changes to the views from update operations made on the base tables, and to our knowledge, this is one of the first works to do so in a local-first DBMS. Users can easily create and refresh the views on command, and applying new changes is done automatically. The project also compares the performance and storage costs of incremental refresh and complete refresh of materialized views in a local-first database.

1.4 Outline

The remainder of the thesis follows the given structure:

- **Background**: Goes into the technical background related to materialized views and CRRs, in addition to work/literature related to this thesis.
- **Design**: Presents the details around significant design choices made and strategies chosen for this work.
- **Implementation**: Describes the implementation of materialized views in SynQLite.
- **Evaluation**: Details the experimental setup, experiments performed, and the results of them.
- **Discussion**: Discusses how the work performs in relation to the requirements and findings of the experiments. It also presents some suggestions for future work.
- **Conclusion**: Summarizes the work and the findings presented in this thesis.

2 Background

This chapter presents some relevant technical background for the thesis regarding solutions and techniques applied in this thesis. This is followed by a selection of related work that has inspired our work and solutions.

2.1 Technical background

2.1.1 Views

Views are a well-established and broadly supported feature of many database management systems [32]. They are virtual tables that allow users to easily access the results of queries as if they were physical tables [1]. They do not store the actual results, only the query definition. This means that they completely rebuild the results each time the view is accessed, and therefore the data is always up to date with the newest underlying changes. Additionally, this means that views are often relatively slow to access, as the entire view has to be recalculated each time it is accessed.

Views have a wide range of applications, such as limiting access to sensitive data, simplifying complex queries, combining data from multiple data sets, and displaying more relevant information to users.

2.1.2 Materialized views

As views must be recomputed on each access, they may cause a large overhead. Recalculating their contents even when no changes have been made, can be a waste of computing resources and time. Materialized views limit this overhead [7], by saving the results of a views query physically, in addition to the query definition itself. This allows for better performance for reads, as the data is already computed and stored. Users can then access the data as if it were a normal table, and no additional computing must be done. For complex and demanding queries involving large volumes of data, this difference may be even greater when comparing standard views and materialized ones.

Materialized views can be used to generate aggregated or summarized data sets for large amounts of data, such as in data warehousing [4]. Large data sets can be queried by a materialized view to create a smaller subset that is faster and more relevant and can be seen as similar to caches. Frequently accessed data can be fetched quickly as the search area is more limited, and any irrelevant data is filtered out.

The drawback of materialized views is that they require storage for their data, in contrast to normal views, as they are stored physically. In addition to this, they need maintenance in order to ensure that the results are fresh, and not out of date as changes in the underlying data sets occur. Depending on the refresh mode, this is not trivial, and the different modes must be considered carefully for the use case. Different methods have different characteristics, benefits, and drawbacks, which are highly dependent on the application of them [23]. Complete refresh for example simply recalculates the entire data set from scratch. Incremental refresh, on the other hand, attempts to limit the amount of data that is calculated, by only refreshing data that has been modified. Many commercial DBMS support materialized views, including Oracle [9], SQL Server [20], PostgreSQL [13], and more.

2.1.3 Local-first software

Local-first software [16] is a set of principles for software that facilitates collaboration and user ownership. It follows the idea that ownership of data and real-time collaboration are not mutually exclusive. In modern cloud apps, the data resides on the server, which has control over it. The user cannot access any data other than through the provider of the cloud service, and as soon as the service is down, the data is inaccessible. "Old fashioned" apps, however, lie on the local disk. Any data is completely accessible without any servers, and the user has full control over it. Local-first software attempts to provide the best of both worlds, with the ownership of a traditional system and the

collaboration of a modern cloud service. Local-first software presents a set of ideals that developers should strive for:

1. **No spinners: your work at your fingertips** - The primary copy of the data is stored on the local device and is therefore accessible immediately. Any synchronization can be done behind the curtains.

2. **Your work is not trapped on one device** - Data is synchronized across devices the user works on. They are not dependent on a single device to access the data.

3. **The network is optional** - The users can access and work on data anytime, even without internet access, as the primary replica lies on the local device.

4. **Seamless collaboration with your colleagues** - Users can collaborate across multiple devices to edit data simultaneously. They should not have to manually send files back and forth, and should not have to worry about conflicts.

5. **The Long Now** - As all data is stored locally first and foremost, the data is saved for a long time and can be accessed offline after the online service eventually shuts down.

6. **Security and privacy by default** - The local device stores only the users' data, and therefore there is no centralized database in the cloud holding everyone's data, which can be vulnerable. The apps may use end-to-end encryption for even more safety, and the servers then only hold encrypted data.

7. **You retain ultimate ownership and control** - The user holds the ownership of the data, not any service provider. This refers not to the legal aspect but to user agency, autonomy, and control.

2.1.4 Conflict-free replicated relations

CRRs (Conflict-free Replicated Relations) is a concept related to CRDTs(conflict-free data types), applied to relational databases [34]. They have been developed to handle some of the issues related to distributed databases and the CAP theorem [3], which states that it is impossible to ensure all three properties at the same time: strong consistency, availability to updates, and tolerance to network partitioning. CRDTs and CRRs provide high availability and partition tolerance with eventual consistency.

In a database replicated across multiple sites, where changes are made independently on its copy of the database, CRRs provide a way of asynchronously merging changes from multiple sites without a centralized coordinator. The results after the merge should be the same as if the database was not distributed, even when changes are disseminated out of order. This is done by keeping some extra metadata for the database such as a record of its history.

2.1.5 SynQLite

SynQLite is an extension of SQLite with CRR support, also called a local-first database [33]. It uses a Python interface to handle synchronization and communication between sites. Replicas of databases are eventually consistent, and any conflicts are resolved automatically without any need for interaction from the user.

Users are able to make a pre-existing database into one with CRR support and interact with the database in a normal SQLite fashion. Metadata will be generated and maintained automatically when the user interacts with the database. It supplies users with the ability to fetch changes from one replica and merge them into another, to synchronize their states. It is based on SQLite and Python and supplies users with a Python interface and shell commands.

2.1.6 Data provenance

Provenance is information about the origin of something. Data provenance tells us something about how a data item has been derived [14]. It holds the source and how the data item has been produced from it, such as the steps taken to produce a result. It is frequently used in distributed systems [21] and DBMS in order to validate and debug, among other applications. Based on the provenance of an item, one can determine the steps needed to create this item, which can then be used to get the quality of the data, retrace errors, and allow automation of updates, among other use cases.

2.1.7 Causal length

A causal length is a natural number that is used to track the current status of a data item and is somewhat related to a version number [34]. When an item is first inserted into a data set, its causal length will be set to 1, and if it is then deleted, the CL will increase to 2. A reinsertion will increment it to 3, and so on. An odd causal length will indicate that the item is currently in the data set, while an even CL indicates that it has been deleted. Subsequent inserts

or deletes on the same item will not affect the causal length, as it will only increase if a delete follows an insert or the other way around. It is based on the idea that insertions and deletions typically happen in turn. This can be used in distributed databases to keep track of a data item's presence, across sites. When synchronizing an item between two sites, the highest causal length wins, as this must be the latest seen insert/delete operation.

2.2 Related work

2.2.1 Augmenting SQLite for local-first software

The authors [33] present an approach for local-first software which aims to allow for offline work as well as collaboration over multiple devices. This is done by applying CRDTs on the relational DBMS SQLite through CRRs.

The work augments the existing SQLite databases with support for synchronization mechanisms between multiple devices. Changes made to the databases can be disseminated to other devices in a decentralized system, and each site merges the changes with its local copy. It includes conflict-resolution techniques to ensure consistency across the sites.

Through their experiments, they show that the approach is able to achieve high levels of performance and scalability while allowing users to collaborate in the offline-first approach.

This work creates the basis for the underlying CRDT and CRR mechanics used in our work. It provides mechanics for version control, creating delta tables, and performing merges of base tables. The views in our work are made on top of this.

2.2.2 Maintaining views incrementally

In the paper [15], the authors describe an approach to maintaining materialized views in database management systems that use an incremental approach to improve efficiency. They discuss the concept of views and challenges related to maintaining the materialized views after changes occur on the underlying tables. One of the main challenges is the high cost of recalculating the entire view each time an underlying change is done.

The incremental approach attempts to accommodate this issue, by only adding the changes made, instead of recalculating everything from scratch. It identifies

the changes made on the base tables and applies them to the view. One of their main contributions is the proposed counting algorithm for tracking view tuples and their base tuples. The counting algorithm is based on counting the number of possible derivations for each view tuple. That means for every base tuple that can be used to construct the view tuple, the count increases. When relevant base tuples are deleted the count decrements, and when it reaches 0 it is deleted from the view as well.

This method works well in a non-distributed environment, but it does not fit well in a distributed one. Different sites may have different base tuples, leading to different counts. The mechanism cannot tell whether or not a change from a site has already been applied from a different site, and does not answer how a view tuple is related to a base tuple. It also does not handle concurrent changes to its tuples, such as when multiple insertions and deletions of the same tuple are performed.

Through a series of experiments, they show the effectiveness of their approach and its ability to efficiently maintain the materialized views. It can handle large data sets and frequent updates, and they conclude that this approach may significantly improve the performance of materialized views in DBMSs.

The work of [15] is closely related to ours. It implements materialized views that use incremental maintenance, which is related to our views. However, it does not take into consideration a distributed setting with replicated databases and local-first properties, which is done in this thesis.

2.2.3 Towards replicated and asynchronous data streams for edge-cloud applications

The paper [22] presents a framework made for the asynchronous transmission of data streams between replicas on edge devices. It uses CRDTs and data provenance on an RDBMS to ensure eventual consistency across devices.

The presented approach allows for querying and sharing data between nodes using materialized views. They are maintained incrementally, which they compare to data streams. To keep track of changes and apply them incrementally to the views they use an idempotent commutative semiring for data provenance in addition to a casual-length lattice to keep the causality of concurrent changes.

The approach supports insert and delete operations on views derived from named conjunctive relational algebra, also referred to as SPJR(select, project, join, and rename). It also supports unions. The implementation is done in

Elixir using Elixir GenServer for the sites and Erlang Term Storage for data storage.

Our work is closely related to this approach, but we use different technologies, such as Python and SynQLite/SQLite instead of Elixir and Erlang Term Storage. Our work does not focus much on reducing provenance expressions to their most simple form as the authors of [22]. Additionally, our approach also supports update operations and not only insert and delete.

3

Approach

The thesis focuses on the approach of the design paradigm presented by [8] for computer science research. This is related to the engineering process for computer science research, and multiple steps are proposed to aid in structuring the research approach. They give a guideline for how the work should be carried out. The other two paradigms presented by [8] are theory and abstraction. The steps for design are:

1. State requirements
2. State specifications
3. Design and implement the system
4. Test the system

By adhering to this design paradigm, we aim to ensure a systematic and structured approach to the development and evaluation of our proposed solution. The design process will be guided by the specific requirements and specifications of our local-first materialized views, incremental maintenance, and the utilization of SQLite and SynQLite. Rigorous testing procedures will be employed to validate the system and assess its performance and usability.

3.1 Development

System development methodologies are formalized steps giving a systematic approach to developing software [11]. They are frameworks that can be used to plan, structure, and organize the development process of a project. Multiple methodologies exist, each with its own strengths and weaknesses.

In this thesis, the development can best be described as close to an agile approach [2], though followed relatively loosely. Through iterative steps, the requirements are determined and reevaluated through regular meetings while solutions are implemented. The method allows for flexibility, which is vital in a project like this, where exploration and research are essential elements. Determining a rigid set of requirements would not be beneficial in our case, as new information is frequently unearthed and may render the requirements and solutions invalid. Goals and requirements are continuously revised as new information comes to light, and adjustments are necessary. New features are added regularly and revisited later when needed.

Tests are made continuously throughout the project as part of the development process. This gives effective feedback on the current state, and how one should proceed further. It helps uncover bugs and unintended behavior as it arises, which may increase the quality of the solution [10]. Most of the tests are functional tests, checking if the contents of the tables comply with the expected results of the operations performed.

One example of the tests used can be seen in 3.1. This test makes a view on the column 'name', and makes an insertion of the tuple *(2, "Alice")* which is then deleted afterward. The view is then refreshed which should result in an empty view. Following this, the tuple *(2, "Alice")* is reinserted as well as the tuple *(3, "Alice")* which will be a duplicate and should not be included in the view. A refresh is then done, and tests check if all metadata and the view results are correct.

Listing 3.1: Test case example

```
function test_reinsert_duplicate(self, crrs)
  self.setup_tbls(crrs)
  query="SELECT DISTINCT name FROM person WHERE id<3;"
  create_view(self.viev_name, query, self.db_local)
  insert(self.crr_remote, self.tbl, (2, "Alice"))
  delete(self.crr_remote, self.tbl, "id", "2")
  refresh_view(self)
  insert(self.crr_remote, self.tbl, (2, "Alice"))
  insert(self.crr_remote, self.tbl, (3, "Alice"))
  refresh_view(self)

  crr_person=self.select_remote(self.tbl)
  cl=self.select_local("view__cl")
  view_table=select_local(self.view_name)
  aug_view=self.select_local(self.view_name+"__aug")

  assert(len(cl)+len(mvtbl)+len(aug) == 6)
  assert(cl[1][0] == crr_person[1][0])
  assert(cl[1][1] == crr_person[1][1])
  assert(cl[1][2] >= crr_person[1][2])
  assert(view_table == [('me',), ('Alice',)])
  assert(aug_view[1][0] == crr_person[1][0])
  assert(aug_view[1][1] >= crr_person[1][2])
  assert(aug_view[1][-1] == crr_person[1][-1])
```

3.2 Work phases

The thesis is worked on in multiple phases and is related to the design approach described earlier. The phases are as follows:

1. Research and design
2. Implementation and testing
3. Testing and evaluation
4. Finalizing and reporting

The first phase consists mostly of designing the solution and performing some

preliminary research. As this thesis is related to materialized views in SQLite, which were implemented in the preceding capstone-project [12] in a non-replicated database, most of the research in this thesis looks into how to solve the problem for a local-first system. The next phase is concerned with the implementation of the designed solution, with testing done along the way. Following this, the next phase is mostly concerned with making experiments and evaluations, in addition to testing. Lastly, the final phase is focused on making some final touches to the project and reporting the findings. These phases are not fixed, as the development follows an iterative approach. The work continuously revisits previous phases, in addition to the following phases. Still, the main focus of each phase is executed as described, and each phase lasts roughly 1 month.

4

Design

The views follow set theory, meaning that there will be no duplicates present in the view [5]. This is equivalent to using the "DISTINCT" keyword [29] in SQL and means that all tuples in the resulting view are unique, even when there are multiple base tuples (points of origin) that can be used to derive the tuples in the view. This requires some additional mechanisms to ensure consistency and correctness. If one of the multiple base tuples that lead to a single view tuple is deleted, one must make sure that the view tuple is not deleted when the other base tuples are present. Additionally, different sites may perform concurrent changes on the same tuples which leads to the need for a mechanism that can handle these cases and provide consistency.

Note that in this thesis we make an important distinction between the terms ’update’ and ’change’. ’Updates’ are referred to only as the update operation in SQL, while ’changes’ encompass any modification of a table, including insertions, updates, and deletions.

This chapter presents the key design choices used for our solution and explains the most crucial parts of the principles applied.

4.1 Provenance expressions

This approach follows a variant of provenance expressions, closely related to the work [22]. The provenance of a tuple is a mathematical expression that explains how the tuple has been derived and is also described in section 2.1.6. A provenance expression [14] consists of references to the base tuples, which are the tuples in the source table(s) used to create each tuple of a view. These references are separated by the operators '$*$' (least upper bound) and '+' (greatest lower bound) which can be viewed as the logical *'and'* and *'or'* operators respectively. Each reference in this case is the globally unique ID of the base tuple(UUID).

Provenance expressions can be considered to be true/false expressions, where a true expression means that the tuple should be included in the view, while a false evaluation means it should be excluded. To consider an element in the expression to be true, the causal length of the elements must be odd, as this means it is not currently deleted. It can therefore be used to derive the view tuple.

Multiple base tuples may be used to derive a view tuple, and how these are used will determine how they are separated. For the operator '+', either one of the tuples separated by it is sufficient to derive the view tuple. An example of this is when duplicate tuples are present in the base tables. As long as one of the duplicates is present, one can procure the view tuple based on these two, and deleting only one will not affect the end result of the view displayed to the user. An example of this can be seen in table 4.1. One can see that both the tuple with a UUID of '*1C21GE*' and '*2TY54S*' can be used to derive the view tuple *'Alice'*. The provenance for *'Alice'* is then '*1C21GE+2TY54S*' before any changes are made. After the tuple with UUID '*2TY54S*' is deleted, the provenance expression changes to only be '*1C21GE*' and the causal length of the deleted tuple is incremented to *2*. Still, *'Alice'* remains in the view as the tuple with UUID '*1C21GE*' is still present.

In the occasion of joins between two base tuples, the references are separated by '$*$' since both of these references are needed to derive the view tuple. If either one of the '$*$'-separated tuples in the expression is missing, the view tuple may not be derived. This means that the tuple with this provenance is not present in the view. In this thesis, we define sub-expressions in a provenance expression, as multiple UUIDs separated by '$*$'. These can be separated by the '+' operator when multiple sub-expressions lead to the same view tuple. Sub-expressions have the same functionality as single provenance elements, meaning that as long as one of the sub-expressions is true, the tuple should be included in the view.

Person			
UUID	Timestamp	Name	Age
1C21GE	1	Alice	20
2TY54S	1	Alice	45
3KLO19	1	Bob	30
4AS4AB	1	Clarice	30
After changes			
1C21GE	1	Alice	20
3KLO19	1	Bob	30
4AS4AB	1	Clarice	30
5L109F	2	Dave	51

Name_augmented		
Provenance	Timestamp	Name
1C21GE+2TY54S	1	Alice
3KLO19	1	Bob
4AS4AB	1	Clarice
After changes		
1C21GE	2	Alice
3KLO1H	1	Bob
4AS4AB	1	Clarice
5L109F	2	Dave

View__CL		
UUID	CL	Timestamp
1C21GEF	1	1
2TY54SA	1	1
3KLO19H	1	1
4AS4ABJ	1	1
After changes		
1C21GEF	1	1
2TY54SA	2	2
3KLO19H	1	1
4AS4ABJ	1	1
5L109F	1	2

View
Name
Alice
Bob
Clarice
After changes
Alice
Bob
Clarice
Dave

Table 4.1: Provenance affected by changes. Red cells will be deleted, blue cells have been changed, and green have been inserted. Changes made are: *(DELETE WHERE ID = '2TY54S';* and *INSERT INTO Person VALUES('DAVE', 51);)* (Note that these provenance expressions do not include an update flag)

To supply users with a view containing only the requested data, one needs to hide the metadata such as provenance expressions. This is done by separating metadata into an augmented layer based on the view, in addition to the view itself. The augmented layer contains an augmented view, a version of the view

which is where the metadata is stored, while the normal view will only contain the attributes specified by its query. The metadata includes the causal length 2.1.7, data provenance, and timestamps, all of which are needed to maintain the views. The augmented layer should not be accessed by the users, only by the underlying mechanisms used by the system. Since it contains data irrelevant to the user, and crucial for the view results, accessing it manually may cause unintended behavior.

4.2 Update flag

Intuitively, when update operations are performed on a single tuple, there is still only one tuple. Previously, this tuple had other values, but they are simply overwritten. In our case, when update operations are performed on a site, there will be two variants for the same tuple: the one with the old values, and the new one. This is because the new tuple is not identical to the old one, and it is therefore considered a new unique row, which has no affiliation with the old one, except for having the same UUID. This leads to update operations being treated as an insert of a new tuple in the augmented view, with an extra property. The old tuple should not exist in the final view, and there must therefore be some mechanism in place to guarantee that it is left out from the view accessed by the user. This is where the update flag comes in.

What separates our provenance approach is that each of the references in the provenance expression will also have an additional operator which we will refer to as the update flag. This update flag is a boolean variable that keeps track of whether or not this reference still holds the attribute values held in that augmented row. This boolean is directly connected to a provenance element, represented with a value in parenthesis after the UUIDs/provenance elements. A provenance expression of '*1C21GE+2TY54S*' will become '*1C21GE(True)+2TY54S(True)*' when adding the update flags. The update flags will initially be set to *true* in all cases. When an update operation takes place and changes the values of the underlying referenced base tuple so that it no longer corresponds to the values held in the augmented view, the update flag will be set to *false*. This indicates that this reference is no longer valid for the current view tuple. At a later point, this referenced tuple may be updated again to be valid once more. The update flag will then be set back to *true*. This is a relatively simple approach that allows tuples to be updated at any point without harming the integrity of the view.

The mechanism uses a last-write-wins approach. A timestamp is tied to the augmented row, which will be checked each time an update operation occurs. On refreshes, the row with the highest timestamp wins, as this is the latest seen

operation on this row. This is similar to if the database was not distributed. This method ensures that update operations can be performed on multiple sites on the same tuples, and the result will still be consistent in the view displayed to the user. When a tuple has been updated out, that is the tuple is no longer valid due to an update operation, the system will still be able to track the tuple. By doing this, it is also able to differentiate between an updated-out tuple and a new duplicate of the old base tuple just inserted. The insertion will not affect the update flags of the existing provenance expression, only add the new provenance element or expression with its initial update flag. Following this, the expression can still be evaluated to be true, even when the old tuple can no longer be used to derive the view tuple.

Take the example seen in table 4.2: There are 3 sites $\mathcal{A}$, $\mathcal{B}$ and C where sites $\mathcal{A}$ and $\mathcal{B}$ each have a replica of the same relation $R(Name, Age)$. C makes a view $V(Name)$. Sites $\mathcal{A}$ and $\mathcal{B}$ each make an update operation at times t_1 and t_2 respectively where $t_1 < t_2$ on the tuple with a UUID of '4AS4AB'. This results in the tuples $r_{\mathcal{A}}('Bob', 30)$ and $r_{\mathcal{B}}('Dave', 30)$. No matter in which order C pulls its updates from sites $\mathcal{A}$ and $\mathcal{B}$, the resulting view tuple will be $v('Dave')$ as the latest update is done on $\mathcal{B}$. If it pulls from $\mathcal{A}$ first, then the tuple will be updated in two steps, and the view tuple will briefly be $v('Bob')$ until the update is pulled from $\mathcal{B}$. If the updates are pulled in the other order, then the view tuple becomes $v('Dave')$ immediately, and the update from $\mathcal{A}$ is simply discarded during the attempted refresh, which is what is illustrated in the tables 4.2.

Additionally in the example tables, one can see how the update flags in the provenance expression for 'Alice' is changed from *true* to *false* for '2TY54S' as this tuple no longer holds the name value 'Alice' after the update operations. The provenance expression in its entirety is still considered to be valid, as the first sub-expression/element '1C21GE(True)' is valid, and the 'Alice' tuple will still be included in the view.

4.3 Evaluating view tuples

When deciding whether or not a tuple in the augmented view should be included in the actual view, multiple evaluations take place. This is required, as the augmented view table is a grow-only set and holds both valid and invalid tuples, and one cannot simply interpret all tuples in the augmented layer to be in the view. Tuples that have been deleted, or are no longer valid due to update operations, will still remain in the augmented view, but should not be included in the view. They are not deleted from the augmented view, as they may be inserted or updated back in at a later point in time, and the CL and provenance

Person (site $\mathcal{B}$)			
UUID	Timestamp	Name	Age
1C21GE	1	Alice	20
2TY54S	1	Alice	45
3KLO19	1	Bob	30
4AS4AB	1	Clarice	30
After updating 'Alice' to 'Bob" and 'Clarice' to 'Dave'			
1C21GE	1	Alice	20
2TY54S	2	Bob	45
3KLO19	1	Bob	30
4AS4AB	3	Dave	30

Person (Site $\mathcal{A}$)			
UUID	Timestamp	Name	Age
1C21GE	1	Alice	20
2TY54S	1	Alice	45
3KLO19	1	Bob	30
4AS4AB	1	Clarice	30
After updating 'Clarice' to 'Bob'			
1C21GE	1	Alice	20
2TY54S	1	Alice	45
3KLO19	1	Bob	30
4AS4AB	2	Bob	30

Name_augmented		
Provenance	Timestamp	Name
1C21GE(True)+2TY54S(True)	1	Alice
3KLO19(True)	1	Bob
4AS4AB(True)	1	Clarice
After refresh		
1C21GE(True)+2TY54S(**False**)	2	Alice
3KLO19(True)+**2TY54S(True)**	2	Bob
4AS4AB(**False**)	3	Clarice
4AS4AB(True)	3	Dave

Name(view)
Alice
Bob
Clarice
After refresh
Alice
Bob
Dave

Table 4.2: Provenance and update flag affected by changes. Red cells will be deleted, blue cells have been changed, and green have been inserted. Changes made are: *UPDATE Person SET Name = 'Bob' WHERE UUID = '2TY54S';* followed by *UPDATE Person SET Name = 'Dave' WHERE UUID = '4AS4AB';*

must be set correctly based on their previous values. For insertions that means incrementing the CL, while for provenance it means setting the update flag(s) to *true* or adding a new element to the expression. By not deleting the excluded tuples, the system will be able to keep correctly tracking changes made on them

at a later point in time, also when considering concurrency.

The provenance expression is split up into sub-expressions, where each of them are separated by a '+' operator. As long as one of these sub-expressions is true, the tuple is valid and should be included in the view. Each of the IDs in the sub-expressions of the augmented table is cross-referenced with the CL table, and we check if the casual length of each of the IDs in the sub-expression is odd or even. This ensures that tuples that have been deleted in the base table, will not be included in the view and that none of the required base tuples in a join are not currently deleted. Additionally, a sub-expression is only true if all of its update flags are set to *true*. If the value is *false*, then this base tuple has been updated into other values, and may no longer be used to derive the view tuple.

4.4 Change propagation

The change propagation works in 5 main steps:

1. Fetching delta
2. Marking updates in the augmented views
    (a) Refresh view table
3. Refreshing the CL-table
4. Refreshing the augmented views
5. Refreshing view table

All of these steps are performed on the view site, but the first step also includes the base site, as it needs to generate the delta file, before the view site can fetch it. For each step in the change propagation procedure, incoming changes are applied, and may or may not be sent further down the chain. An overview of the steps involved in the change propagation can be seen in figure 4.1. These steps essentially calculate how the base changes affect the data in the current step, and filter out any changes that do not affect the result. Through this process, changes will move iteratively through the steps, calculating and applying changes to metadata, and finally determining and applying the changes to the views. Some changes done on base tables will not result in any changes in the view at all, but they may cause changes in the augmented layer. This could be a new provenance expression or incremented causal length. Each step

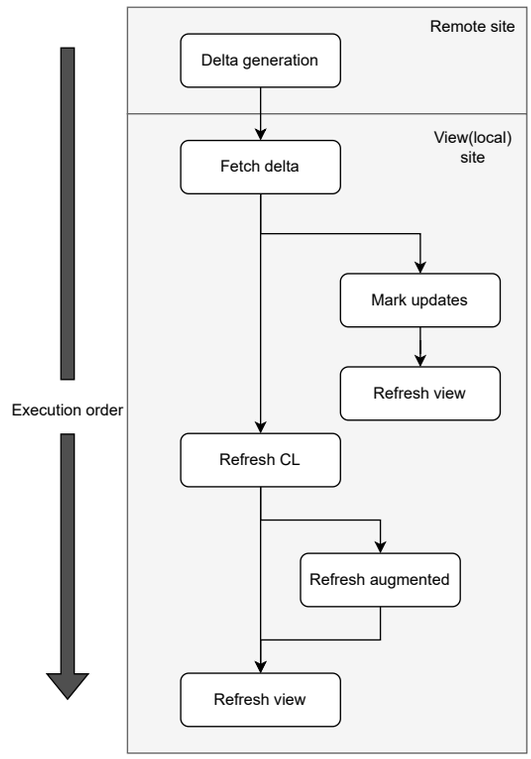

Figure 4.1: Change propagation

down the chain is generally more narrow in what it allows to "pass through", as changes applied to one step may not cause any more changes further down the chain.

Any change in the base table will be seen in the delta table, a set of all changes that have occurred since a given reference point. Changes on a site will be tracked by SynQLite and the history will be kept in a table. When site $\mathcal{A}$ wants to refresh its views from the base tables on site $\mathcal{B}$, it will fetch all changes made on $\mathcal{B}$ since its last refresh, by retrieving the delta table calculated by $\mathcal{B}$.

The first step is to set the update flags in the provenance expressions if any update operations have been performed on $\mathcal{B}$. If the tuples in the delta table satisfy the "WHERE"-condition for the view, they can be eligible to be applied to the CL table. The changes applied to the CL table will then be sent further down to the augmented tables, where they may cause a change to be applied if it affects the provenance expression of an existing tuple or causes a new tuple to be inserted. The changes in the augmented tables are then compared with the causal lengths in the CL table, and may or may not cause a change to move further down the chain and onto the view. If a duplicate is affected, the change

may not be applied to the view, and updating causal lengths of tuples where a deciding element has an update flag of 0 will also cause no change.

It is not a given that the set of changes to be applied to the next step is smaller than the one in the previous step, however. A single delete operation on the base table may lead to a single increment of causal length in the CL table, which may cause no changes in the augmented table. Still, multiple tuples may be deleted in the view, assuming they were dependent on the base tuple deleted. The point still remains, as there must be a change in one of the previous connected steps in figure 4.1, for changes to be applied to the current step.

5 Implementation

The chapter describes the details of how the solution is implemented and the most important aspects of the implementation.

5.1 Materialized views

As SQLite has no support for materialized views, one needs to implement a way of physically storing the results of a query. This is done by using normal tables. The results of a given query are calculated through a series of steps, where the data produced is saved in a table. There are no mechanisms supplied by SQLite that allow automatically merging new results with previous ones, which is a fundamental part of materialized views. One must therefore implement these mechanisms manually using insert, update and delete operations. These will be constructed based on the queries provided when creating the view.

To ensure uniqueness in the view, a unique index [25] is made on the view, which also may increase query performance in some cases on that view table. There is also no support for a dynamic "insert or update" operation of new data. One can however emulate this by dividing it into two separate operations, where one inserts when possible and updates when it is not. Alternatively using "insert or replace" [27]. As long as the view tables have the option to refresh, they are relatively similar to materialized views in their functionality.

5.2 Table structure

For each view, there are two tables specific to that view: one that holds the results of the views query, and one augmented table. The augmented table holds the view's attributes in addition to a provenance expression and a timestamp of the last seen change. When the results of a view are calculated, a query will be performed on the augmented version, where the provenance and causal length will be evaluated to determine if each tuple should be included in the view or not. In addition to holding the attributes and metadata, the augmented table is used to abstract and limit to only the most relevant data for the user. Note the difference between an augmented table, which is a single table specific for the view, and the augmented layer, which includes the augmented view table and the CL table.

The causal length of each tuple is stored in another table, along with the tuple UUID and timestamp of the last seen change. The causal length is a single integer, complying with the stated behavior of a casual length attribute described in 2.1.7. The UUID is the same as for the underlying CRR mechanisms and is copied from the CRR tables or delta tables generated by SynQLite. These UUIDs can be found in the provenance expressions of the augmented views and allows checking causal length for each element in a provenance expression. There is only one CL table on each site and they keep track of the tuples of all views on that site. This table is also a grow-only set and therefore includes tuples that are no longer valid. Tuples that are deleted from the base tables will have their casual length set to an even number, but the CL-tuple will still remain in the table. By keeping the causal length in a table separate from the augmented view, one can limit the number of columns needed in the augmented view. There is only a need for one reference point to the causal length for each of the base tuples, and keeping it in a separate table accommodates this.

To keep track of all views on a site, a table holds each view's name, query, and uniquely generated id. These tables only track the views of the site it resides on and will have no information concerning the other sites' views. The data in this table is needed to retrieve the query used to create a view, as this query is used to perform refreshes. This table will be referred to as the view query table. Figure 5.1 displays these tables and how they are connected.

It is important to note that the base tables themselves are not maintained on the view site unless there is a join statement. Any changes on the base tables will be seen in their entirety in the delta tables and can be produced from the information in them. After the changes from the delta tables have been applied, they are discarded and will be fetched again when needed. By only keeping the delta tables temporarily, one limits the amount of storage space needed and the size of the search area.

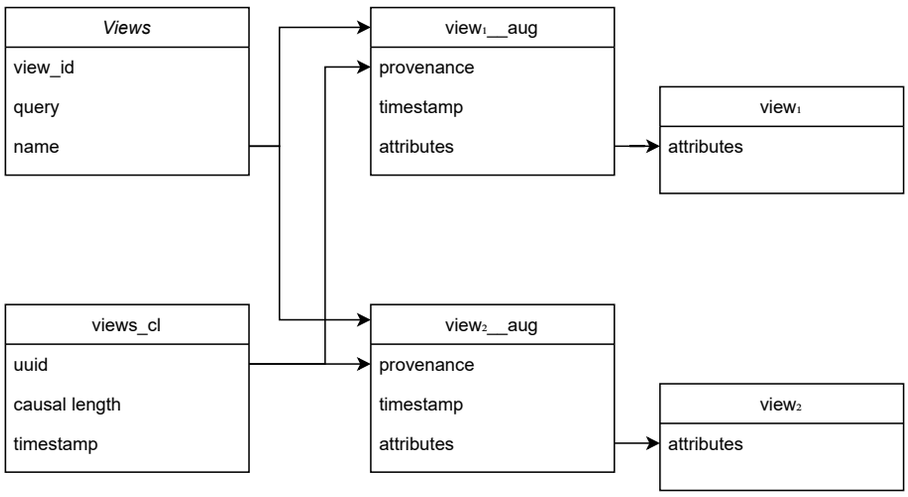

Figure 5.1: Table structure

Timestamps are included in the augmented views and the CL tables. The ones in the CL tables are used when considering if insert and delete operations are newer than the last seen change. The timestamp in the augmented views is used when considering update operations on the base tuples, to check if they are newer than the last seen update operation on each augmented tuple. Using only the timestamp in the augmented views, one would not get a sufficiently fine-grained approach to evaluate every single causal length, as concurrent inserts/deletes may be done on only a few base tuples. Using only the timestamps in the CL table, one would have to compare all of the timestamps related to the unique base tuples of the provenance expressions and the incoming update timestamp. This would also be insufficient, as an update operation is dependent on comparing the values of the view attributes when applying the change, and must therefore be tied together.

5.3 Creating views

Materialized views are made by passing a standard SQL query as an argument to the Python interface along with the views name. The query will be parsed by a simple query parser and saved in the query table. A table for the view is made, along with the augmented view table. If this is the first view on this site, the CL table and view query table are also created. The contents of the view will initially be based on the current local data at the site of the view. If there is no local data, then the view is empty until it is refreshed from another site. The view can then be accessed like any other table in SQLite, and will only contain relevant information, as all metadata is held in other tables. Views

can be refreshed by using the Python interface or shell commands, which will refresh all views located on the site, based on changes made on a given remote site. One does not need a reference to each individual view to do this.

5.4 Select

Selection criteria used when creating a view will be saved as part of the query in the view query table. They will be used all throughout the change propagation already from the CL-refresh stage to filter out any tuples not part of the view. As the delta tables from the pull site include all its data, this is an important step in ensuring that tuples that do not satisfy the criteria are not tracked by the view systems, and will not affect the views in any way. To be able to verify that all tuples in a view satisfy the selection criteria, any attributes in the selection must also be in the project part of the view. If the attribute is not part of the view, one cannot check if a tuple is updated to a value that does not satisfy the selection.

5.5 Project

Project is also supported by this implementation. It works by choosing a subset of columns from a base table to be part of the view. Any chosen column in the view query will be saved in the views query. Any time a refresh is performed, the query is fetched, and with it, the project part as well. The column's names will be the same in the view as that of the base table, and its origin is specified by the prefix, eg. "person.name" when specifying that the name attribute should come from the person table. These prefixes are rewritten through the refresh execution to match the query's use case. For example, when fetching the incoming delta values, the prefix will be replaced and result in "delta.delta__person.name", which will target the desired column from the delta table.

5.6 Joins

The implementation supports inner joins on two tables. Outer joins and self-joins are left for future work. In order to produce the view tuples from a join statement, one needs the entire left and right-hand side of the join. This means that both entire base tables must be kept available and up to date on refreshes. This is why one of the first steps in change propagation with joins is to merge

the delta with the local base tables. This allows for rewriting the query to join a delta table with a local base table and be able to produce every new tuple that may be derived from the incoming delta. For each of the tables in the join, we rewrite it to become the delta version of it, joined against the other tables as local replicas. If one were to only perform the query on the delta tables joined against the other delta tables, one would not get all possible combinations of tuples satisfying the query, as the delta tables only hold the most recently altered tuples. This would result in a non-complete result which would not contain all the tuples it would in a non-distributed setting.

Take the example seen in table 5.1: The relations *vio(vio_id, business_id, date, type)* and *insp(insp_id, business_id, score, date)* are replicated on sites $\mathcal{A}$ and $\mathcal{B}$, and the tables have one tuple each, *vio('2UT4RR', 7602, 20171212, 100001)* and *insp('35IVN1', 7602, 55, 20160105)*. A view *V(business_id, type, score)* is made on site $\mathcal{A}$ using the query: "*SELECT DISTINCT vio.business_id, type, score FROM vio JOIN insp ON vio.business_id = insp.business_id.)*". The view will initially have one tuple: *v(7602, 100001, 55)*. Site $\mathcal{B}$ makes an insertion of tuple *vio*('1P8ANN', 2447, 20160923, 34111) which is followed by site $\mathcal{A}$ refreshing the view from $\mathcal{B}$ at time t_2. Site $\mathcal{B}$ creates a delta table δ_{vio} containing all changes on *vio* which means *vio('1P8ANN', 2447, 20160923, 34111)*. This is pulled by $\mathcal{A}$ and the refresh starts. This will still result in a view with only one tuple, as there are no tuples in $insp$ with a matching business_id for vio to be joined on. Still, the local replica of vio will be merged with delta, resulting in *vio('1P8ANN', 2447, 20160923, 34111)* in vio on $\mathcal{A}$ as well. After this, site $\mathcal{B}$ inserts *insp('8H2LAS', 2447, 89, 20170314)* into $insp$ and site $\mathcal{A}$ refreshes V from $\mathcal{B}$ again at time t_3. This time the delta table δ_{insp} contains a tuple that can be joined with vio. $\mathcal{A}$ rewrites its query to become "*SELECT DISTINCT vio.business_id, type, score FROM vio JOIN insp on vio.business_id* = δ_{insp}*.business_id*" which will produce the results *v(2447, 34111, 89)* which is stored in the view.

By performing the joins on one delta table and one local replica, one ensures that only the new tuples may be produced from the incoming changes (delta), and one limits the number of rows to compare. This is in contrast to if one were to join the local tables with each other after applying the delta to the local tables, which also includes old already seen tuples and could decrease performance.

5.7 Query parser

The query parser is relatively simple and is used to define the different parts of a query. It takes in the query as a string and detects SQL keywords such as "SELECT", "FROM", and "WHERE" in order to decide what the other words in the

Violations			
vio_id	business_id	date	vio_type
2UT4RR	7602	20171212	100001
After insertion at t_2			
2UT4RR	7602	20171212	100001
1P8ANN	2447	20160822	103116

Inspections			
insp_id	business_id	score	date
35IVN1	7602	55	20160105
After insertion at t_3			
35IVN1	7602	55	20160105
8H2LAS	2447	89	20170314

View_augmented				
provenance	timestamp	business_id	type	score
KJH887(True)*B75KA2(True)	1	7602	100001	55
After refresh at t_2				
KJH887(True)*B75KA2(True)	1	7602	100001	55
After refresh at t_3				
KJH887(True)*B75KA2(True)	1	7602	100001	55
IY541Z(True)*PX38MM(True)	3	2447	103116	89

View__CL		
UUID	CL	timestamp
KJH887	1	1
B75KA2	1	1
After refresh at t_2		
KJH887	1	1
B75KA2	1	1
After refresh at t_3		
KJH887	1	1
B75KA2	1	1
IY541Z	1	2
PX38MM	1	3

View		
business_id	type	score
7602	100001	55
After refresh at t_2		
7602	100001	55
After refresh t_3		
7602	100001	55
2447	103116	89

Table 5.1: Provenance and update flag affected by changes. Green cells indicate newly inserted tuples. Changes made are: *UPDATE Person SET Name = 'Bob' WHERE UUID = '2TY54S';* followed by *UPDATE Person SET Name = 'Dave' WHERE UUID = '4AS4AB';*

query mean. Using this parser, one can split up the query into more manageable parts, and reconstruct it to fit the current use case. The implementation reuses as much as possible from the input query string, to allow for greater freedom

and a larger degree of customization. It is also far from trivial to mimic the exact behavior of SQLite and SynQLite in Python, which is why all of the data management and calculations are done by writing or rewriting the query into SQLite commands and executing them through the SQLite interface for Python. The query will be reconstructed and used multiple times throughout the refresh cycle. Additionally, queries may be written in many different ways and still be valid and equivalent to other queries, reinforcing the need for a flexible parser.

5.8 Incremental refresh

The implementation of incremental refresh is more complicated than a complete refresh, as there is more metadata to maintain and more specific actions to be taken. The following listing shows the main steps involved in the execution of incremental refresh:

- Get views
- Get delta file
- Merge delta file if join
- Attach delta
- Mark updates
- Modify query parameters
- Insert or replace CL
- Reverse
- Insert row into augmented view
- Insert row into view
- Update rows in the augmented view
- Insert or delete into view

The first step is to fetch the views and initialize them as Python objects. Next, the delta file is pulled from the given pull site and merged with the local CRR

files if there is a join in any of the views. If there are no join statements, then the local CRR files remain untouched. Then the delta file is attached to the local database file and the merging of the view starts. Pseudo code for the general merging of the delta with views can be seen in listing 5.1.

Listing 5.1: Algorithm for overall incremental refresh

```
function merge_delta_views(cursor, views)
for view in views:
    for table in view.delta_tbls:
        view.mark_updates(cursor, table)

altered_rows = []
for view in views:
    for table in view.delta_tbls:
        if join in view.query:
            view.modify_query_parameters(table)

        inserted = view.insert_to_cl()

        if join in view.query and is_delta(table):
            join_table = get_joined_table(table)
            inserted += reverse_ins_to_cl(join_table)

        altered_rows += inserted

for view in views:
    for table in view.tables:
        if join in view.query:
            view.modify_query_parameters(table)

        for row in altered_rows:
            inserted = view.insert_to_aug_view(row)
            view.insert_to_view(row, inserted)

            view.update_augmented_view(row)
            view.refresh_view(row)
```

Since updated rows are similar to inserted rows, except for having a non-unique UUID, they must first be evaluated to determine the update flag of the old row values. A query is generated to identify updated rows. This is done by joining the delta rows with the CL table on UUIDs and joining again with the

augmented view where the UUID is in the provenance expression and checking that the incoming timestamp is higher while satisfying the WHERE condition. Additionally, the CL must be the same in the CL table and the delta table, to rule out any insert or delete operations. The result will be a set of tuples that have had their attributes changed and already exist in the view or augmented view, which is only the case for tuples modified by an update operation.

Each of the updated tuples will then be inspected. To find the old tuples in the view corresponding to the updated one, a query is made to retrieve all provenance expressions that include the updated tuples UUID with an update flag of *false* and where the attribute values are equal to the updated tuple. This will get the provenance of tuples where the updated tuple has previously been updated out, that is updated so the tuple is no longer valid. These provenances will then be updated to have their update flag set to *true*, as to indicate that it has been changed back and is now valid again. The next step is to set the update flags for invalid tuples. This starts by fetching the provenances of tuples where the updated tuples UUID is in the provenance expression with an update flag of *true*. These are from tuples that have been valid up to this point. Then an update operation sets the update flag to *false* on these provenances, where the attribute values are not equal to the values of the updated tuples. At this point, all provenance expressions currently in the view should have the correct update flag values. Before moving on to the other delta rows, a refresh of the view is done to apply the changes to the view itself. The pseudo-code for the update operations function *mark_updates* can be seen in listing 5.2.

Listing 5.2: Algorithm for setting the update flags based on incoming updates

```
function mark_updates(cursor, table)
  fields = get_column_names(tbl)
  updates = self.get_updates(table, fields)

  for row in updates:
      old_tuples = get_invalid_tuples(row)
      for row in old_tuples:
          set_update_flag(row, 1)

      new_tuples = get_valid_tuples(row)
      for row in new_tuples:
          outgoing_tuples = set_update_flag(row, 0)

          for out_row in outgoing_tuples:
              self.refresh_view(out_row)
```

Now inserted and deleted rows can be considered. First, a query is constructed to insert or replace the new tuple in the CL table. The data to be inserted comes from a selection on the delta tables joined with the delta history, adding the "WHERE" criteria, and ensuring that the tuple is not already present in the table with a larger timestamp. Any insertions resulting from this query mean that an insert or delete operation has been done on the source site, and there might be a delete or insert operation to be performed on the view. It is not guaranteed however, as insertions and deletions of duplicates may not lead to any changes in the view. If no insertions are done to the CL table, this delta tuple should not cause any more changes to propagate further in the system.

The next step is to consider the augmented views, and any tuples inserted in the previous step will be looked at. The changes done on the CL table may result in either a new tuple to be inserted into the augmented view and/or an update on the provenance expression of already existing tuples. This is done in two steps: insert and update. An SQL expression is generated, which inserts or ignores new augmented tuples with the same UUID as the ones inserted into the CL table. There is a unique index on the view attributes, and any attempted insertion of duplicate tuples will be ignored. Each successfully inserted tuple will then be attempted to be inserted into the actual view, or ignored if it is a duplicate. Next, the update expression is generated. It generates the new provenance element based on the tuples inserted into the CL table and adds it to the existing provenance expressions, where the view attributes are the same. This ensures that new base tuples can be considered when looking at how a view tuple is derived.

Lastly, the view table is incrementally refreshed based on the tuples inserted into the CL table. Listing 5.3 shows pseudo-code for this function. First, all provenance expressions containing the CL tuples UUID are selected. Each of these provenance expressions is evaluated and each individual UUID in them is compared with the causal length in the CL table. The evaluation determines whether or not the tuple should be in the view. For it to be in the view two conditions need to be satisfied. As long as one of the sub-expressions gotten by splitting the entire expression on "+" satisfies these conditions, the tuple will be in the view. The conditions are:

1. The casual length of all provenance elements must be odd

2. The update flag of all provenance elements must be "(True)"

If the provenance expression of a tuple is valid, an insertion of that view tuple is performed, and it will be ignored if it is a duplicate. If the provenance expression is even, a delete operation is done instead, where the view attributes are equal to the augmented tuple evaluated. After these steps are done, the view should

be up to date on all changes fetched from the pull site and applied.

Listing 5.3: Algorithm for applying changes to the view

```
function refresh_view(cursor, table)
  relevant_rows = get_aug_rows(provenance(row))

  for row in relevant_rows:
      is_prov_valid = evaluate_provenance(row)

      if is_prov_valid:
          tuple_values = format_values(row, self.query)
          insert_to_view(tuple_values)
      else:
          delete_from_view(row)
```

5.9 Complete refresh

For the purpose of performance comparisons, the implementation also supports complete refresh, which is a refresh where all data of a view is recalculated from scratch each time it is refreshed. The main steps of a complete refresh are:

- Get views
- Get delta file
- Merge delta file
- Drop metadata tables
- Recreate view from scratch

The first step is getting all the views from the view query table. These will then be initialized by parsing the queries and defining objects of the view class to easily access them. Next, we fetch the delta file from a site specified by the user. This uses the underlying SynQLite methods of generating a delta file based on recent changes on that site. The next step is merging the delta file with the local replicas of the base tables. This is done using the supplied SynQLite methods that handle the merging and results in up-to-date base tables in the local replica. When this is done the data needed to remake the views is available through

the local replicas. The next step is dropping all metadata tables related to the views so they can be remade later. Now all view tables and metadata tables are remade from scratch, using the methods used for initializing new views. This uses all available data from the local base table replicas, and the views will be up-to-date.

6 Evaluation

This chapter describes the experiments performed and presents the results of them. Some findings are presented along with the reasoning behind their occurrence. The experiments focus on performance and storage costs for the implemented views.

6.1 Experimental setup

All experiments and evaluations are executed on the same single computer with the following key specifications:

- OS: Ubuntu 22.04.1
- CPU: Intel i7-11700 @ 2.50 GHz (8 cores)
- Memory: 16GiB DDR4 3200 MHz
- Disk: Kioxia M.2 SSD 512 GB 33MHz

They are performed using Python 3.10.6 and SQLite version 3.37.2 with the Python SQLite interface version 2.6.0.

The experiments are not run in a distributed setting but simulate it using mul-

Experiment tables	
Violations	Inspections
vio_id INTEGER PRIMARY KEY business_id TEXT date TEXT violationtypeid INTEGER risk_category TEXT description TEXT	insp_id INTEGER PRIMARY KEY business_id TEXT date TEXT score INTEGER type TEXT

Table 6.1: Base tables queried by the views in the experiments

tiple directories and SQLite database files. For instance, a "local" folder and a "remote" folder each contain a database file, and are both located on the same machine. They will not be synchronized in any way other than that of the experiments' refreshes. The SQLite files are instantiated as CRR databases and a series of operations are executed on each database individually. The experiments then treat the "remote" folder as if it were located on another machine, but will not have to go through the network to pull its changes. This simulated approach should be sufficient to test the functionality of the solution, as the focus of this thesis is to look into using SynQLite for local-first views and developing a method for handling update operations on the base tables. Actual networking is not required to see if the proposed solution is viable.

Experiments are run using the same base relation on each run, and the order of operations will also be the same each time. 2000 operations are done on the remote table one at a time, followed by refreshing the view for each operation. This is also repeated with the number of operations per refresh set to 100 instead of one at a time, still resulting in 2000 operations in total. This is done to see how the number of operations per refresh affects the performance. Each experiment is repeated a number of times, and the averages are calculated and used for all plots, tables, and comparisons. There are three main experiment categories, one for insertions, one for updates, and one for deletions. These operations are done on the "remote" database file and a refresh is done on the "local" database file pulling changes from the remote one. The refresh is done incrementally and completely, and the results are compared. The three types of experiments are each done on a project view and a join view which will be described further.

Experiments also consider two variants of the same experiments, shuffling and non-shuffling modes. Shuffling the order of execution means that the update operations and delete operations are executed in a random order, while non-shuffling means that they are executed in the same order as they were inserted.

Views	
MV_project	MV_join
business_id	business_id
date	date
violationtypeid	type
	score

Table 6.2: The two views used in the experiment

6.1.1 Project experiment

The experiment starts by creating a project view as "SELECT DISTINCT businesss_id, date, violationtypeid FROM violations". This will initially be empty, as there is no data in the local violations table. Two variants of the experiments are run, one where one insertion is done for each refresh, and one where 100 insertions are done for each refresh. The time used to perform the refresh is taken and added to the set of results. After all iterations are done, the experiment is repeated 5 times, and the average times are then used for the plots. The results of the time used to refresh the views based on insertions done on a remote site can be seen in figure 6.1.

For the update and the delete experiment, all views, tables, and data from the insertion experiment are reused, to have some data to work with. Update operations are performed on the date values and simply increment the date by 1. What value is changed has no impact on the results, what matters is that some arbitrary value is updated, to be defined as a change. The operations are executed in the same fashion as for inserts: doing 1 update operation per refresh, then repeating the experiments with 100 operations per refresh, and timing how long the refresh takes. When it comes to delete operations, the views, tables, and data from the insert experiment are reused again. A delete operation deletes tuples from the base table based on the vio_id and is executed in the same pattern as updates and insertions. The results of the update experiment can be seen in figure 6.2 and deletions can be seen in figure 6.3.

6.1.2 Join experiment

Experiments on views using joins are made from the query "SELECT DISTINCT vio.business_id, vio.date, type, score FROM vio JOIN insp on vio.business_id = insp.businesss_id". This view is initially empty since the local inspections(insp) and violations(vio) tables are empty. 2000 insertions are done on each table at the remote site and the view is refreshed, pulling the remote changes.

The insertions on each base table are not guaranteed to match on business_id, meaning that they may not result in complete tuples that can be included in the view. The same goes for the delete and the update experiment. Not all operations will affect the actual view, only the underlying CRR tables. These experiments are performed in the same way as for the project experiment, where the results of the insertion experiment are reused for the update and delete experiments. It is also performed with 1 operation on each table per refresh, and 100. The results for the time used to refresh a view using joins and performing inserts, updates, and deletes on a remote site can be seen in figure 6.4, figure 6.5, and figure 6.6 respectively.

6.1.3 Select experiment

There is no explicit experiment for the select operation. This is because its behavior is not dramatically different from the project one, as the only difference is including the "WHERE <*selection criteria*>" in all queries used in the metadata. It will still produce similar results when included in the join experiment and the project experiment.

6.1.4 Disk usage experiment

The experiments just presented also measure the disk usage of the database. For each refresh, the size of all tables is measured and summed up, which also includes all metadata tables. The measurement is done using an SQLite query on the *dbstat* table [31] which stores information regarding the disk space of the tables in the database file. SQLite only tracks disk usage in terms of pages used by each table, meaning there are only rough estimates of the actual disk usage. It should still be sufficient to see the trends and characteristics of the system, however. Size measurements are done for both the join experiment and the project experiment.

6.2 Results

For each experiment, there is a plot made. Each plot consists of 4 subplots, where two of them use no shuffling of execution order, and two use a randomly shuffled order. Each of these two pairs has one plot for 1 operation per refresh and one for 100 operations per refresh. The plots also include the median, in addition to the averages, to get a clearer picture with less prominent spikes.

6.2.1 Project performance

When looking at the graph for insertions and a project view with 1 operation per refresh in figure 6.1, one can see that the incremental approach is faster than the complete refresh approach, in every step except for a few spikes. The complete approach graph grows exponentially with respect to the data size in contrast to the incremental one, which remains approximately constant throughout the entire graph. This is to be expected, as the complete approach recalculates the entire set of results on each refresh, in addition to the metadata tables needed to calculate the view. The incremental approach on the other hand, only calculates and applies the current incoming changes.

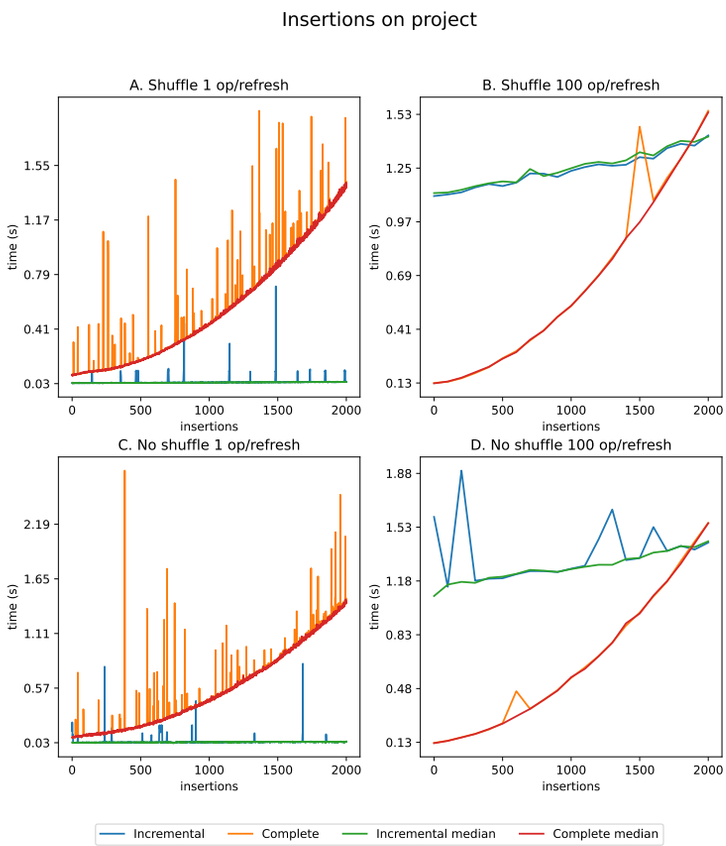

Figure 6.1: Time taken to refresh following a number of insertions on a project view. **A**: Shuffled operation order and 1 insertion per refresh. **B**: Shuffle operation order and 100 insertions per refresh. **C**: No shuffle of operation order and 1 insertion per refresh. **D**: No shuffle of operation order and 100 insertions per refresh.

The number of changes made is very low on each refresh, meaning that there

is a large difference in the number of rows calculated for each approach. For a view with n number of tuples, the number of view tuples to calculate on an incremental refresh is 1, while for the complete approach, the number is n. Additionally, this is the case for the metadata tables, as the incremental refresh only has to add 1 new tuple to the existing CL table and 1 to the augmented view for each insert, while the complete refresh recalculates these too from scratch.

When the operations per refresh are increased to 100 it reveals a slight linear increase in time taken to refresh incrementally while the complete refresh still grows exponentially. The incremental refresh is slower than the complete refresh, to begin with. At about 1900 tuples inserted, the incremental refresh starts becoming faster than the complete. The reasoning behind this pattern may still be attributed to the fact that the incremental refresh calculates only the incoming changes and not the entire table as the complete refresh does, but both seem to be affected by the size of the tables, only less so for the incremental refresh. When there are more tuples (100) to calculate on each refresh, the incremental approach naturally uses more time than for 1 operation per refresh.

For updates on the project view in figure 6.2, one can see that the incremental approach performs better than the complete one at 1 update per refresh. The incremental approach graph is relatively constant throughout the measurements. Similarly, the complete approach is close to constant, but at a higher time taken. Since the incremental refresh mode treats newly updated rows as new ones and keeps the old values in the augmented view tables, the amount of data stored increases. This does, however, not seem to have a large effect on the time needed to execute a refresh. The complete refresh mode does not save the old values of the updated tuples and therefore has about the same amount of data to consider on each refresh, which naturally leads to a relatively constant execution time. Again, on each refresh, the incremental approach only has to calculate the results for 1 tuple, while the complete approach has to calculate n tuples for a table of size n which causes a much larger time demand.

When the number of updates per refresh is set to 100, the complete refresh becomes faster than the incremental one. The complete approach still uses the same amount of time, as there is the same amount of tuples to calculate as for 1 operation per refresh. The incremental refresh however has to perform more calculations in comparison to when performing 1 operation per refresh. Naturally, this causes a longer refresh time. One can also see that the incremental refresh graph increases slightly as the number of updates goes up both for the shuffled and non-shuffled order. This can be attributed to the fact that in update operations of random order, more pages may be used by SQLite, causing larger search areas and slower execution. This will be expanded upon further

in the disk usage section 6.2.3.

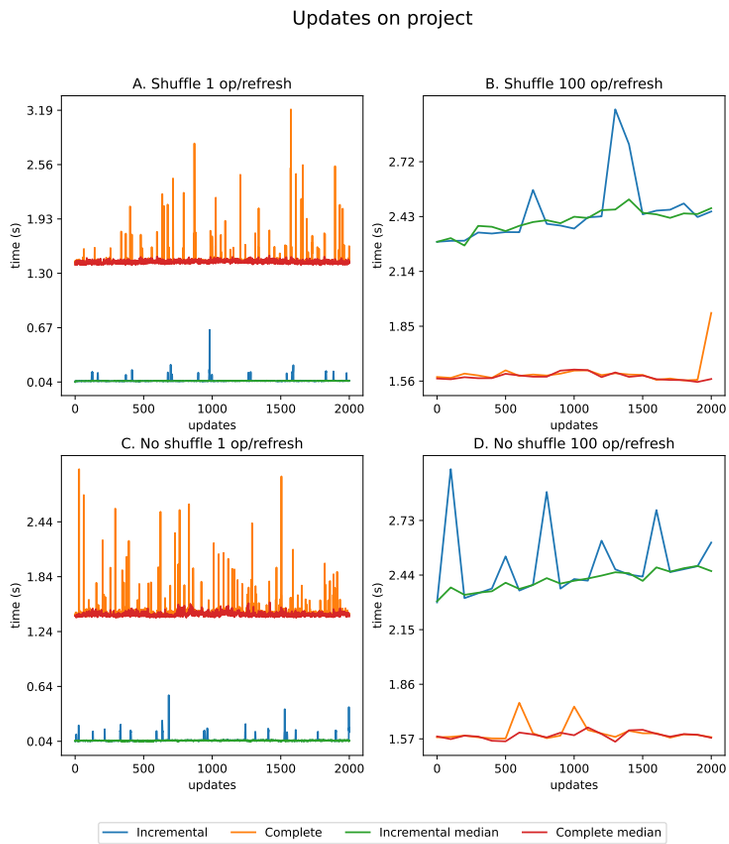

Figure 6.2: Time taken to refresh following a number of updates on a project view. **A**: Shuffled operation order and 1 update per refresh. **B**: Shuffle operation order and 100 updates per refresh. **C**: No shuffle of operation order and 1 update per refresh. **D**: No shuffle of operation order and 100 updates per refresh.

When it comes to 1 deletion per refresh and the project view, the incremental approach graph is constant, while the complete refresh graph decreases roughly linearly. This can be seen in figure 6.3. As the number of tuples in the view shrinks, so does the time taken to perform a complete refresh, as it is directly dependent on the number of rows with regard to execution time. Fewer tuples mean a faster refresh. The incremental refresh is faster overall, which again can be accredited to the fact that the incremental refresh only calculates the result of 1 changed tuple at a time, while the complete refresh calculates the entire view again.

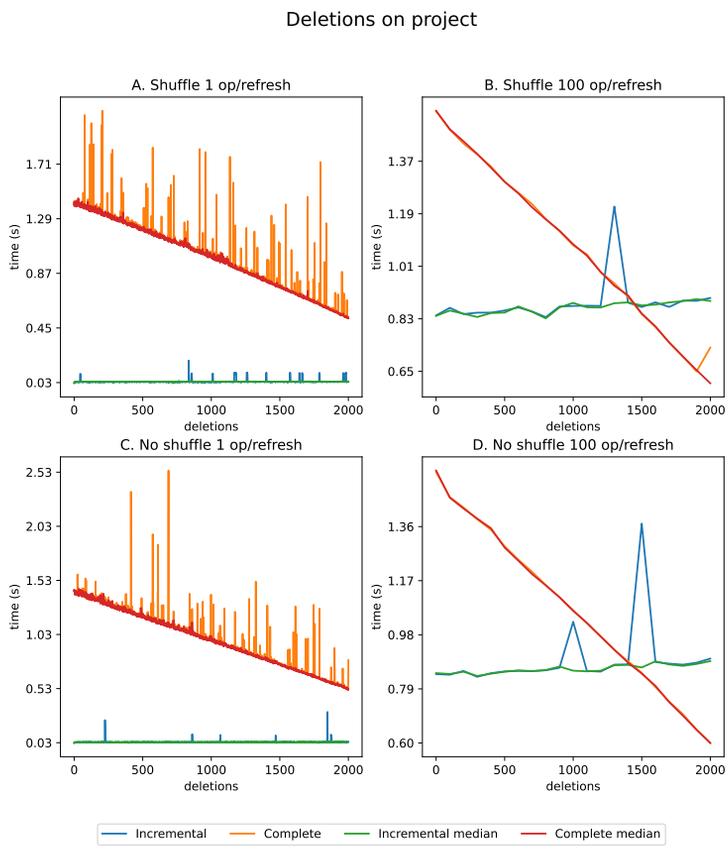

Figure 6.3: Time taken to refresh following a number of deletions on a project view. **A**: Shuffled operation order and 1 deletion per refresh. **B**: Shuffle operation order and 100 deletions per refresh. **C**: No shuffle of operation order and 1 deletion per refresh. **D**: No shuffle of operation order and 100 deletions per refresh.

When the number of deletions per refresh is set to 100, the results change slightly. The incremental refresh now has a very slight linear increase in time taken in relation to the number of tuples. As old tuples are not deleted from the metadata tables, only the view, the amount of data grows even when tuples are deleted. This negatively affects the execution time and the graphs move toward an intersection at around 1400 tuples deleted. The complete refresh takes longer to execute, to begin with, but becomes faster than the incremental refresh when the number of tuples in the view moves toward 0. This is to be expected as the complete refresh does not keep "old" metadata and has less to calculate.

6.2.2 Join performance

When looking at joins and 1 insertion per refresh, one can see from the graphs 6.4 that the incremental refresh mode performs better than the complete refresh mode in general. For insertions the time required to refresh increases for both approaches, as is natural when the amount of data grows. The complete refresh increases exponentially, while the incremental refresh has a very slight linear increase. The time increase for the incremental refresh may however be accredited to a slightly different reason, though still related to the increased size of data. When insertions are made on the underlying base tuples, the value of the attribute in the join condition may not match. This means that the left and right side tuples inserted cannot be joined, but they will be added to the local base table replicas. As more tuples are inserted over time, the chances of two tuples satisfying the join criteria increase, and they may produce a view tuple. This leads to a more frequent insertion to the view as more insertions are made, and the graph has a higher frequency of spikes in time used the higher the number of insertions. Still, the increase in time taken is also affected by the number of tuples considered, as there is a larger search space when querying the tables.

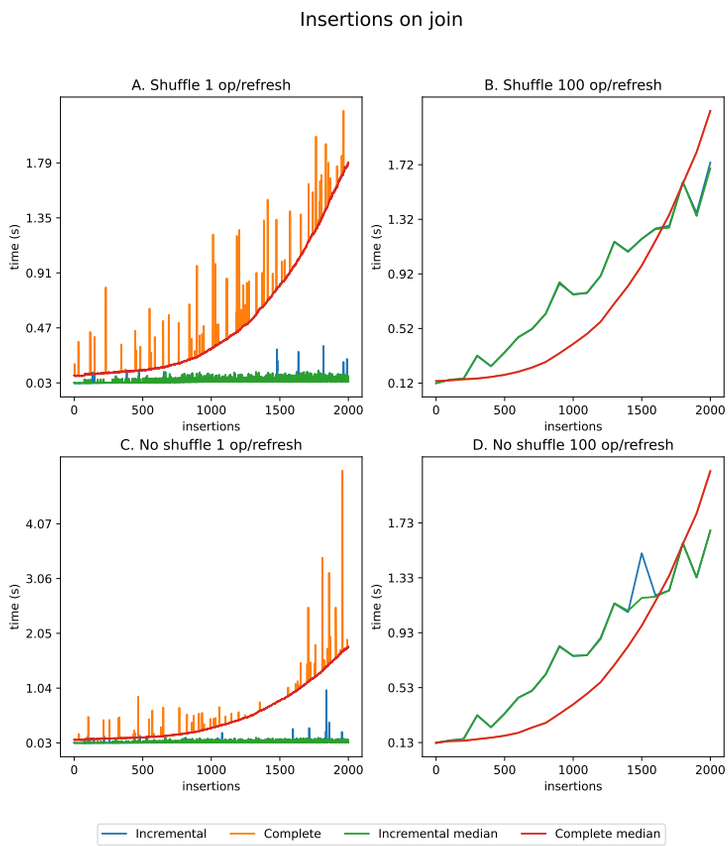

Figure 6.4: Time taken to refresh following a number of insertions on a join view. **A**: Shuffled operation order and 1 insertion per refresh. **B**: Shuffle operation order and 100 insertions per refresh. **C**: No shuffle of operation order and 1 insertion per refresh. **D**: No shuffle of operation order and 100 insertions per refresh.

The complete approach seems to be more stable, as there is only one more tuple in each base table to take into accord when calculating the results, and the time needed to execute the refresh should not be dramatically different from the last refresh. The graph grows exponentially in relation to the number of operations previously performed similar to that of the project experiment. The incremental approach varies more between each refresh, but still generally uses less time than complete refreshes, even when considering the spikes. Looking at the graphs for 100 insertions per refresh, there are some changes. The complete refresh still grows exponentially, but the incremental refresh seems to have a more significant linear increase in comparison to 1 insertion per refresh. The difference in time needed when comparing shuffled order and non-shuffled order of insertion are insignificant.

For updates on the join view with 1 update per refresh in figure 6.5, the incremental approach is still relatively constant. Not every update on the base table leads to a change in the view, as the base tuple affected by the update may not be included in the view. However, the chance of affecting the view seems to be close to constant. This is because the update operations do not affect the join condition, and the number of tuples in the view is constant throughout the experiment. The columns in the join condition are intentionally left out of the update operation, to keep the number of view tuples the same throughout the experiment. The odds of getting a "hit" on a tuple that is also in the view is the same for any number of updates performed, and there will be no increase in spike frequency as there is for the insertions.

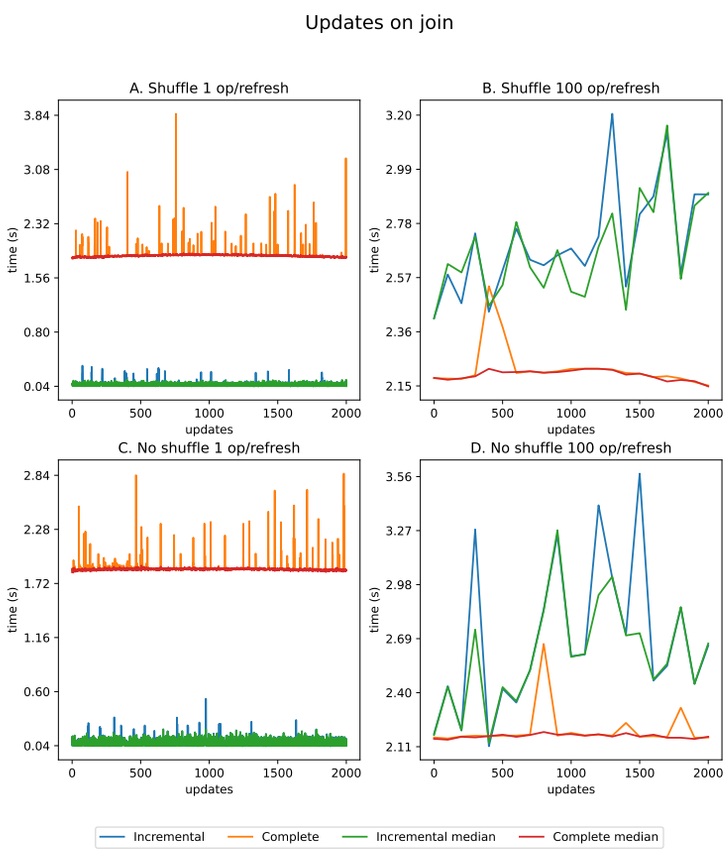

Figure 6.5: Time taken to refresh following a number of updates on a join view. **A**: Shuffled operation order and 1 update per refresh. **B**: Shuffle operation order and 100 updates per refresh. **C**: No shuffle of operation order and 1 update per refresh. **D**: No shuffle of operation order and 100 updates per refresh.

The graph for complete refresh is very slightly concave, but close to constant,

and is slower than the incremental refresh for 1 update per refresh. It does not matter if an update operation affects a view tuple or not, the number of tuples to calculate for the view remains the same. When the number of updates per refresh is increased to 100, the incremental refresh starts increasing in relation to the number of tuples and becomes slower than the complete refresh, which remains at roughly the same time needed. It also becomes very erratic, which may be because of the random chance of an update leading to a change in the view. The increase in time can be accredited to the fact that the old values of the updated tuples remain in the metadata tables, and the search space for the queries increases. The incremental approach naturally has worse scaling here as it stores the old values of the updated tuple in the metadata tables in addition to the new ones.

The delete on join view experiment seen in figure 6.6 shows that the incremental refresh is again close to constant, while the complete refresh graph decreases approximately linearly. One could expect that the incremental refresh would have some more variations depending on the number of deletions, as is the case for insertions, but it is even more consistent in terms of spikes. As each view tuple is dependent on two base tuples, and deleting only one would result in a delete in the view, the first deletions should have a higher chance of affecting the view than the last deletions. This can not be seen in the graph, however.

The decrease in time needed for the complete refresh is to be expected for the join view as well as the project. The amount of data to calculate decreases and the more deletes are performed on the base tables, the fewer computations need to be performed. As the time needed to execute a refresh is highly dependent on the number of tuples, this is to be expected. For the experiment with 100 deletions per refresh, the complete refresh performs roughly the same, but the graph for incremental refresh starts decreasing when the number of total deletions increases. This shows that the increased amount of metadata in comparison to the complete refresh, does not cause a larger overhead as the number of deletions increases, and the lower number of view tuples lead to faster refreshes here too.

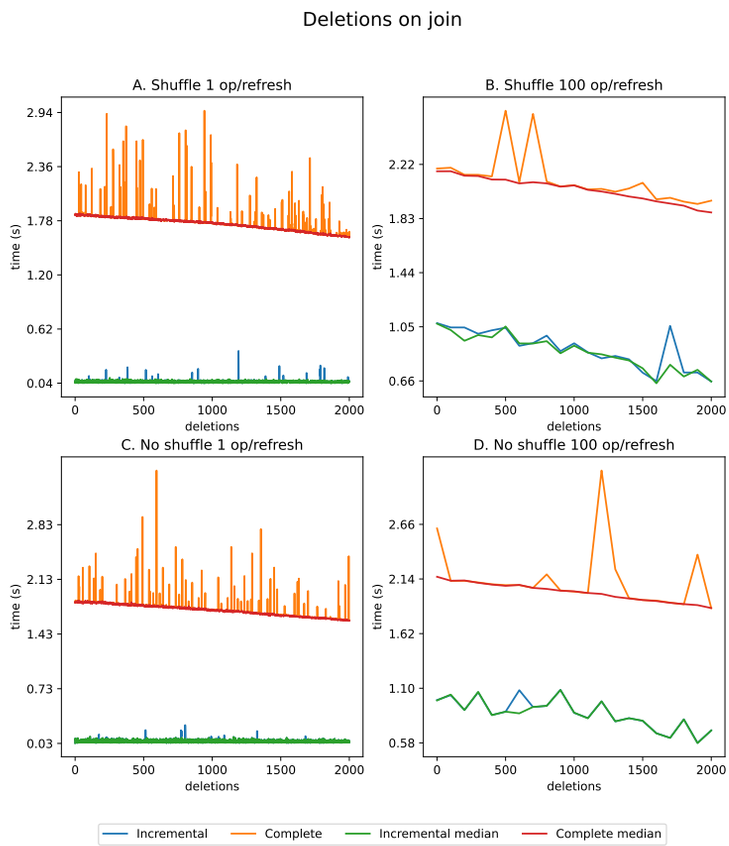

Figure 6.6: Time taken to refresh following a number of deletions on a join view. **A**: Shuffled operation order and 1 deletion per refresh. **B**: Shuffle operation order and 100 deletions per refresh. **C**: No shuffle of operation order and 1 deletion per refresh. **D**: No shuffle of operation order and 100 deletions per refresh.

6.2.3 Disk usage

The experiment results in figure 6.7 show that disk usage goes up as insertions are made. This is to be expected, as disk usage directly correlates to the number of tuples in the tables. Both the incremental and the complete approach increase linearly in terms of space used in relation to the number of tuples inserted in total. The complete refresh increases faster than the complete refresh, however, and the results are the same for all four variants of the experiment. In other words, shuffling and number of insertions per refresh does not matter for insertions on the project view in terms of disk usage.

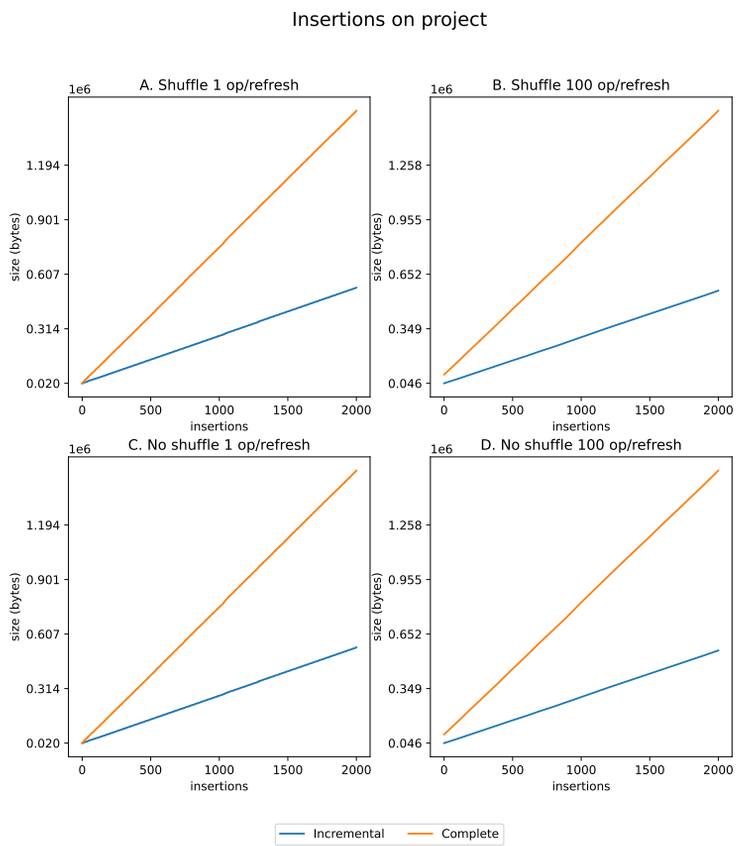

Figure 6.7: Size of tables following a number of insertions on a project view. **A**: Shuffled operation order and 1 insertion per refresh. **B**: Shuffle operation order and 100 insertions per refresh. **C**: No shuffle of operation order and 1 insertion per refresh. **D**: No shuffle of operation order and 100 insertions per refresh.

As previously mentioned, the update operations also lead to an increase in the number of tuples for the incremental refresh mode, and therefore a higher disk usage as the total number of updates increases. These results can be seen in 6.8. This pattern does not emerge in the update experiment for the complete refresh mode, as it does not store as much metadata. It only holds the relevant augmented view rows, and not the old updated values as it does for the incremental refresh. This leads to constant disk usage for the complete approach, while the incremental approach has a linear increase and a generally higher space requirement when there is no shuffling of the order of updates. When shuffling is turned on, the updates occur in a random order and not in the same order as the insertions. This leads to a slightly exponential graph which flattens towards the end (last updates) for the incremental refresh. The complete refresh is also curved with a concave shape. The reason for the curve is that

a random order of updates means that more may pages are used due to the fragmentation of data. As long as a single tuple resides on a page, that page is considered used, and the database takes up more space. Randomly picking a tuple to update leads to a concentration of more pages used around the middle of the experiment, as at this point there is a larger chance for pages to be partially filled up. After the spike, the first pages used start being freed, while new pages are used. Updating in the same order as the tuples were inserted means that the first pages used will be freed when all the tuples on that page have been updated. An example of this is illustrated in table 6.3. Initially, all tuples lie ordered neatly into full pages which is a number of 3 pages. After 6 updates one can see that performing the updates in order leads to the first two pages being freed, and page 3 and 4 is used instead. Still, the database only uses 3 pages. When the order is shuffled randomly, and 6 updates are performed, the result is more scattered. Tuple 1, 5, and 6 have not been updated, and remain on their original page. The rest have been updated, and have been placed into pages 3 and 4. Note that page 2 has been freed. In total, the number of pages used when shuffling the order becomes 4. If all 9 updates were executed, the number of pages used would be 3 for both the non-shuffled and shuffled approach. In general, the incremental refresh uses less space than the complete refresh. This is natural as the complete refresh also maintains the local replicas of the base tables.

	Initial	**No shuffle**	**Shuffle**
Page 0	Tuple 1 Tuple 2 Tuple 3		Tuple 1
Page 1	Tuple 4 Tuple 5 Tuple 6		Tuple 5 Tuple 6
Page 2	Tuple 7 Tuple 8 Tuple 9	Tuple 7 Tuple 8 Tuple 9	
Page 3		Tuple 1 Tuple 2 Tuple 3	Tuple 3 Tuple 2 Tuple 4
Page 4		Tuple 4 Tuple 5 Tuple 6	Tuple 8 Tuple 9 Tuple 7

Table 6.3: Example of page usage before and during update execution.

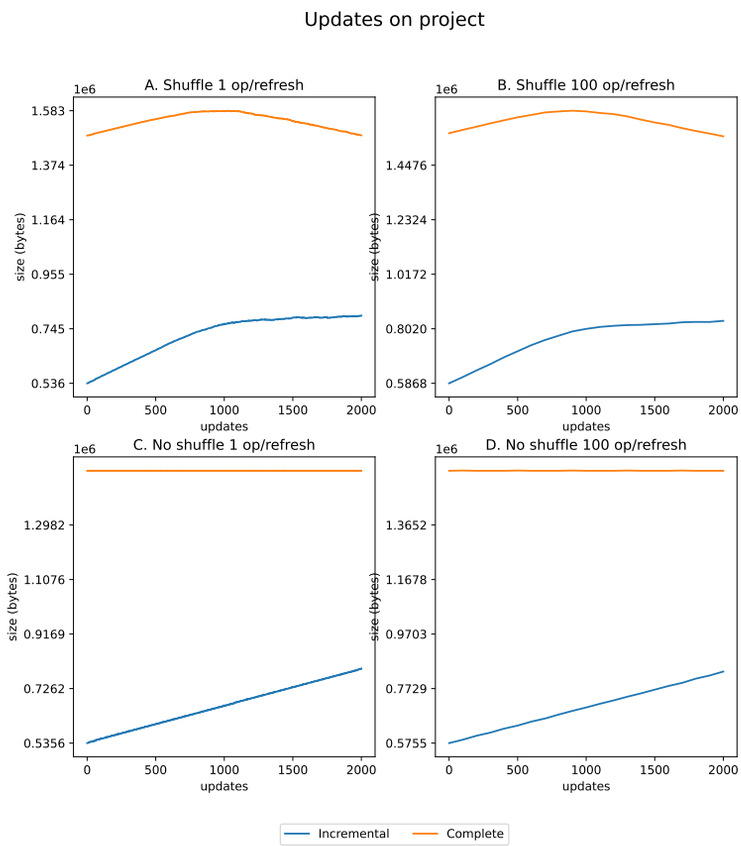

Figure 6.8: Size of tables following a number of updates on a project view. **A**: Shuffled operation order and 1 update per refresh. **B**: Shuffle operation order and 100 updates per refresh. **C**: No shuffle of operation order and 1 update per refresh. **D**: No shuffle of operation order and 100 updates per refresh.

For deletions in the project experiment seen in figure 6.9, the number of tuples in the view goes down as the number of deletions increases, while the number of metadata tuples does not. Overall, it leads to linearly less storage needed as the total number of deletes goes up, for both refresh modes when there is no shuffling. The incremental refresh mode requires less space than the complete refresh mode for the project views in general. When shuffling is turned on, one can see similar trends to the update experiment, that is both graphs become concave. The reasoning is the same as for the update experiments, as a delete leads to an update operation of the CL table, which increases page usage. When all operations are done, the first used pages start being freed.

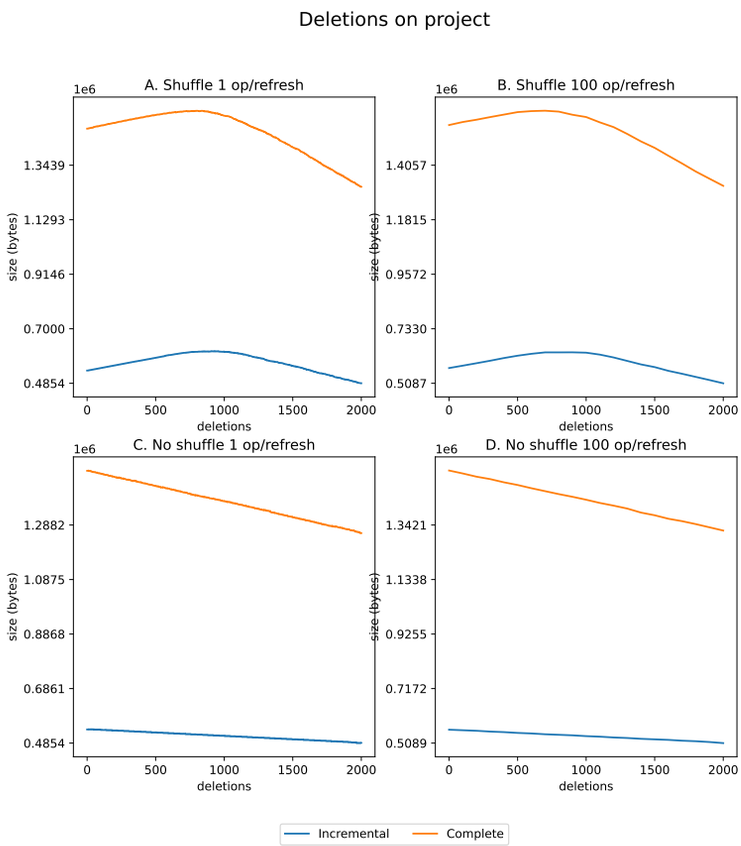

Figure 6.9: Size of tables following a number of deletions on a project view. **A**: Shuffled operation order and 1 deletion per refresh. **B**: Shuffle operation order and 100 deletions per refresh. **C**: No shuffle of operation order and 1 deletion per refresh. **D**: No shuffle of operation order and 100 deletions per refresh.

When looking at the results for joins in figure 6.10, there is generally more data stored. As already mentioned, for join views using incremental refresh, one also maintains local replicas of the CRR tables, which means that there is overall more disk usage. The complete refresh mode views already maintain the CRR tables locally, and the difference between disk usage for the project and join views is therefore not as drastic as for the incremental views. When looking at insertions, the disk usage scales roughly linearly, but slightly exponential, and shuffling and the number of insertions per refresh have no significant impact.

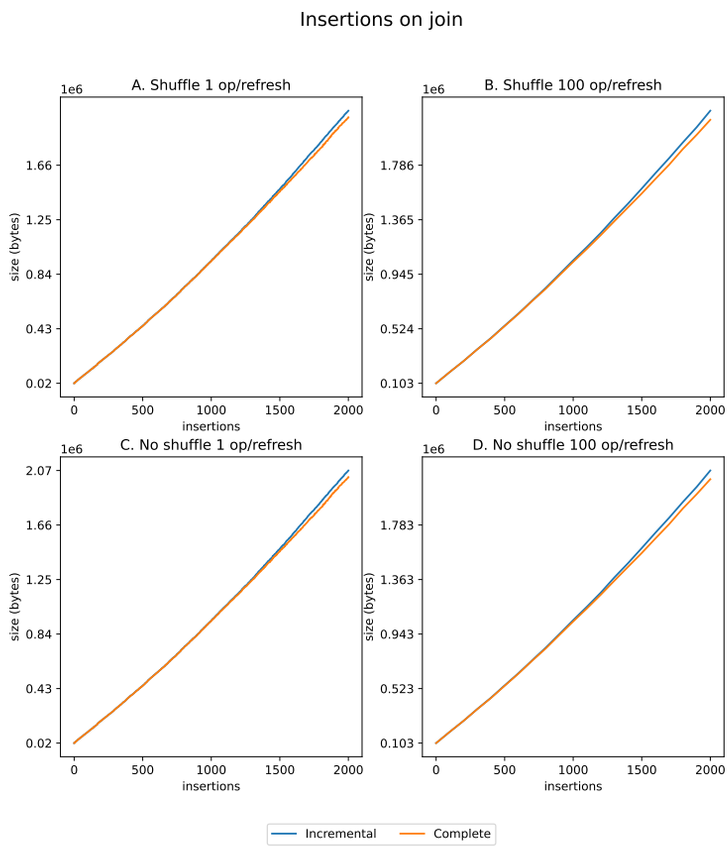

Figure 6.10: Size of tables following a number of insertions on a join view. **A**: Shuffled operation order and 1 insertion per refresh. **B**: Shuffle operation order and 100 insertions per refresh. **C**: No shuffle of operation order and 1 insertion per refresh. **D**: No shuffle of operation order and 100 insertions per refresh.

For update operations on the join experiment in figure 6.11, the complete approach uses a roughly constant amount of space, while the incremental approach increases with the number of total updates when there is no shuffling. Both graphs have a more curved graph when shuffling is turned on. This is the same case as for the project experiment, but the graphs are even more curved, which may occur because there is more data in general. The incremental approach uses more space than the complete approach, which can be attributed to the fact that the local replicas of the base tables are also maintained, which demands more space.

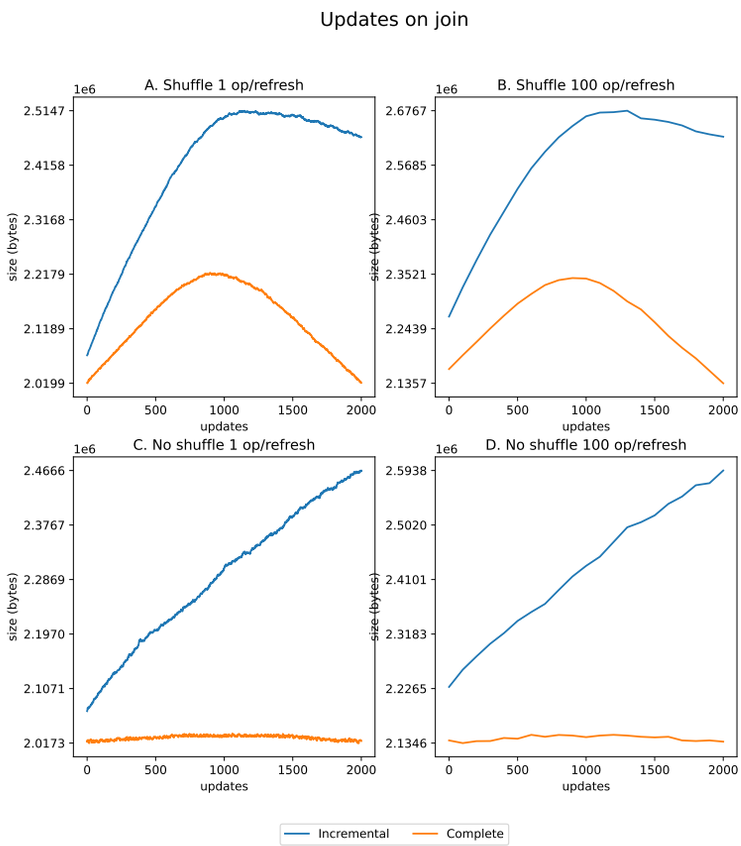

Figure 6.11: Size of tables following a number of updates on a join view. **A**: Shuffled operation order and 1 update per refresh. **B**: Shuffle operation order and 100 updates per refresh. **C**: No shuffle of operation order and 1 update per refresh. **D**: No shuffle of operation order and 100 updates per refresh.

When considering delete operations the amount of disk usage decreases linearly with regard to the amount of deletes performed for both refresh modes when there is no shuffling. These results are shown in figure 6.12. This is as expected as disk usage is dependent on the number of tuples stored. The incremental refresh mode uses more space overall, as there is more metadata to maintain in comparison to the complete refresh mode. When shuffling is turned on, the graphs become concave once again, which may be because the deletions only lead to less space used when a page is completely unused by SQLite. The top of the curve lies a little to the left of the middle, as to begin with pages are not empty yet, and there are more pages used when tuples are updated.

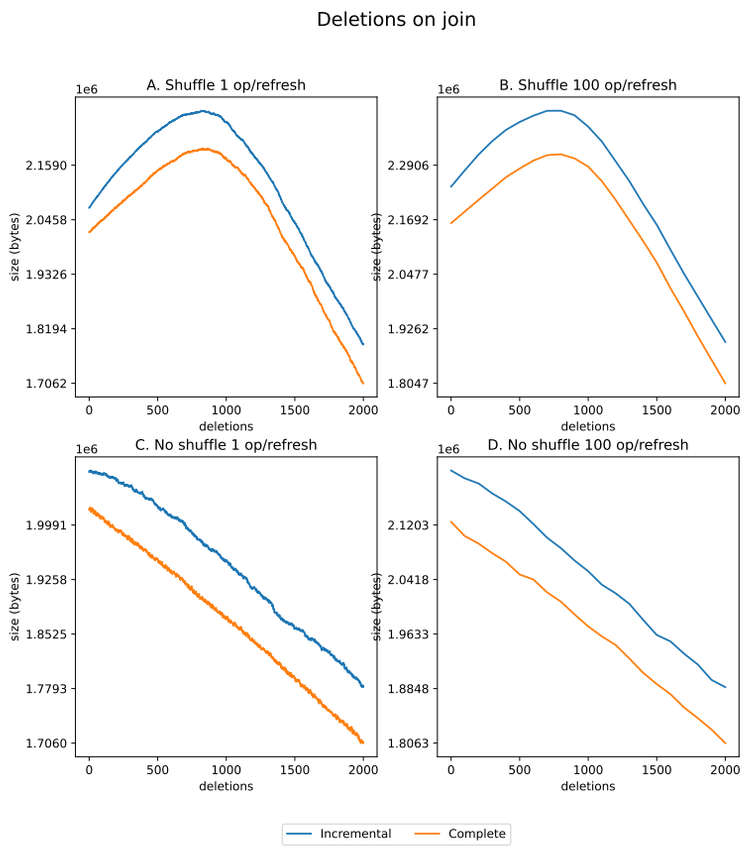

Figure 6.12: Size of tables following a number of deletions on a join view. **A**: Shuffled operation order and 1 deletion per refresh. **B**: Shuffle operation order and 100 deletions per refresh. **C**: No shuffle of operation order and 1 deletion per refresh. **D**: No shuffle of operation order and 100 deletions per refresh.

6.2.4 Operations per refresh

For the project view, one can see that the incremental refresh is generally faster, or scales better than the complete refresh in most experiments. The incremental refresh graph is close to constant in almost every case when the number of operations per refresh is set to 1. In contrast, the complete refresh has some more variety to it, depending on the type of operations performed. When the number of operations per refresh is set to 100 the results become less lopsided and reveal that the incremental approach does not outperform the complete approach in every case. In some cases, as for insertions, the incremental approach grows more linearly.

The incremental refresh shows an increase in time taken as the number of operations per refresh goes up, while the complete refresh uses roughly the same amount of time, as it already recomputes all the view contents and metadata. It also shows that at a certain point in the number of operations per refresh where the complete refresh becomes faster and more suited than the incremental refresh. One must therefore carefully consider the use case of the views, and choose a refresh mode that is best suited. If frequent refreshes are done, and the number of operations performed is small, the incremental approach is generally faster. If the refreshes are performed more rarely and a larger number of operations are done, then the complete refresh may be better.

6.2.5 Spikes

For all the graphs of time used, one can see relatively frequent and large spikes. These are more frequent for the complete refresh graphs than for the incremental refresh. There are multiple possible reasons for this phenomenon, and the biggest suspect is memory management and disk access. SQLite accesses databases using caches, and each page is read into the cache memory when accessed from the disk [30]. This is supported by the fact that the complete refresh generally handles more data, and performs larger calculations (recalculating the entire view and metadata), which may have a larger possibility of running out of cache space. This requires the system to fetch data from the database file which is stated to be an order of magnitude slower than accessing main memory [30]. The memory may also be affected by other processes running on the machine during the experiment execution, making these spikes more unpredictable.

7

Discussion

This chapter discusses how the work done compared against the stated requirements in section 1.1. It details some of the implications of the evaluation and how the work translates to real-world use. Additionally, some quirks and potential issues of the solution and implementation are discussed. Some suggestions for future work are also provided.

7.1 Materialized views

The implementation uses normal SQLite tables to represent views and their metadata, which are stored physically on the disk and will be persistent and available offline. The view's contents are calculated during explicit refresh calls in a deferred approach, and any changes made on the base tables will not be seen in the view until the user has given a refresh command. The contents of the view can then be accessed directly like any normal SQLite table, without any additional calculations or overhead. This allows for fast reads for the user and any demanding calculations can be performed at non-critical times to avoid any delays in the user's experience. This is in contrast to normal (non-materialized) views, where the results of the view must be calculated each time it is accessed. This slows down read times, and demanding or large queries can take a relatively long time to compute before the data can be read.

One issue with using normal SQLite tables for materialized views is that they

allow for interactions that are normally not available for views, such as writing to it. Normal SQLite views are read-only, but there are no mechanisms in place to limit the users to only allow reads on materialized views. One could make use of triggers and an additional table or column to implement a write lock. This can be a simple boolean whose value determines whether or not a write will be allowed or not but is not done in this thesis.

7.2 Local-first views

One aspect of the thesis is to provide views that follow the properties of local-first software, where the main aspect is to allow the use of the views while the device is offline and synchronize the views when an online connection is available at a later time. The implemented solution should support this in theory, but extensive testing on a real-world online system has not been done. The views can be accessed offline at any time without any limitations regarding their offline status. Any data held in the view when online, will remain when going offline, as there are no mechanisms that are dependent on an online connection, except for the refresh. It does require an online connection to fetch the changes made to the base tables, but as soon as that data set has been retrieved, the refresh execution occurs offline, and the data can be accessed regardless of any connection. This is also the case when creating the view initially, as there must be some data available to base the view contents upon.

Generating and fetching delta is done using the functionality provided by SynQLite. As SynQLite aims to support local-first properties, it already provides a good base for local-first views to build upon. It takes care of tracking changes on the base tables, which are used to get base table changes, that can be further examined by the view mechanisms to get the relevant changes. SynQLite also provides synchronization mechanisms that are used to merge incoming base table changes with the local base tables in the complete refresh and in incremental refreshes where a join is present in the views query.

One issue with using local-first views and SynQLite is that the synchronization of base tables interferes with the views if one wants to use CRR tables located at the same site as the views. Since SynQLite lets sites track the other sites and when they have last fetched delta, a synchronization of the base tables may cause a view refresh not to get all changes since its last refresh. This is because the timestamp of the last pull will be higher than when the view was refreshed, and the site that generates the delta assumes that any changes done before the last pull has been seen by the requesting site. One must therefore choose to either use local-first views or CRR-tables on a certain site or perform synchronizations and refreshes at the same time. The complete refresh does

this already, and by calling the refresh command, syncing of base tables is done too. The incremental refresh does not do this unless there is a join in its query. One could relatively easily implement a method for syncing base tables too when doing an incremental refresh.

7.3 Query support

The implementation allows for creating and maintaining views using the most fundamental query techniques, select, project, and join. These may be combined or used without the others and grants a basis for further development of functionality, such as adding aggregations, CTEs, outer joins, etc. The basic SPJ queries should be sufficient to provide some insight into the qualities and potential of the techniques and technologies applied in this solution. As the queries written by the user are parsed and repurposed for all stages of the refresh, the user can write normal SQLite queries in a natural way for creating the views, and the syntax is the same as for standard SQLite.

There are some limitations to what the system can handle, however, other than limiting to SPJ queries. SQLite and SQL in general offer a lot of functionalities, and supporting all of them in our implementation is far from trivial. Using aliases for column names is one example. Aliases can provide more readability and allow for obfuscation for protection and abstraction purposes and are commonly used when working with views. The views also follow set semantics and are intended to not include duplicates. Because of this, the user must include the "DISTINCT" keyword when initializing a view.

The incremental refresh treats queries containing "JOIN" very differently from queries that do not. As long as this keyword is included in the query, the incremental refresh will have to perform some additional string modifications before constructing the queries used in the refresh and also synchronizing the local version of the base tables. The impact of this can be seen all throughout the experiments performed, where the main aspects are that join views require more storage and are slower to refresh.

7.4 Refresh modes

The main refresh mode of this thesis is incremental refresh, but it does also support complete refresh. They both make use of the same metadata and share some techniques used to calculate the view results. There are however some differences, and it is not intended to use both refresh modes interchangeably

on the same site. Upon creation of a view, one should already have in mind what refresh mode should be used. Using both may cause some unintended behavior, and one should carefully consider the characteristics of the modes for the intended use case of the views. As seen in the experiments, the incremental mode is generally faster than the complete, but it costs more storage space when using joins. If the refreshes are frequent and the number of changes is relatively small each time, then the choice should usually be incremental refresh. If refreshes are performed more rarely, and the changes to the base tables are large, then the complete refresh mode may perform better. When storage space is very limited, and the user intends to use a join view then the incremental refresh mode may take up too much space to be used, and the complete approach could be a better fit. This is especially true when the number of operations is high in comparison to the number of refreshes.

One key difference between the two is that the complete refresh does not consider update flags. This is because the results are completely recalculated each time they are refreshed, and the old value of an updated tuple is overwritten from the start. Therefore it does not need to keep track of the old tuples and maintain the update flags.

7.5 Autoincrement

Autoincrement is a mechanism common in relational DBMS that will automatically increment the integer primary key of tuples when inserting, to avoid reusing primary keys and ensure uniqueness. Using the autoincrement mechanism in local-first views and SynQLite may however cause some errors. It may override the manually inserted primary key, and cause inconsistent results across sites. If different sites insert a tuple with the same values, autoincrement may choose a different primary key for the same tuple, as the data this value is based upon can be different on the different sites. It may also cause errors even when there is only one site performing insertions, because of this override.

The results of the view will then vary depending on what value the system chooses for the primary key, which again depends on the order of insertions. This is an especially large issue when performing joins based on the primary key. Additionally, if a tuple is deleted and later inserted again, the primary key is not guaranteed to be the same, even though the tuple is intended to be the same, which may also cause some unintended results.

SQLite automatically applies autoincrement to primary key columns of type integer without any option to turn it off. One must therefore be careful when

creating views, and try to avoid using integer primary keys for the join condition. There are some ways to go around this problem, such as using a different column type or replacing the primary key attribute with a unique constraint on the column instead.

7.6 Multi site merging

When new tuples are inserted into a base table in SynQLite, a unique UUID is generated. The UUID is a randomized value and is not deterministic. This means that tuples with the same values inserted at two different sites will have a different UUID. SynQLite uses the UUID of tuples to identify them, and as the two tuples have different UUIDs, they will be treated as two different tuples when synchronized. This may cause unintended behavior when refreshing a view from different sites. This is however an issue that has been somewhat accommodated in SynQLite as it uses a mapping to identify tuples across sites. This is not made use of in the current local-first view implementation, as it is out of scope. If the sites do not attempt to insert the exact same tuple, the solution is able to refresh from any available site.

7.7 String comparison

SQLite does not enforce data types of columns, instead, it uses affinities [26]. These give a hint about the datatype in this column but do not ensure that the values are of the correct type. When creating a table in SQLite, the standard column affinity is set to be of type "BLOB" if no affinity is specified. When comparing column values, such as in a query, any column two columns can be compared regardless of affinity. This may not always behave as expected, such as when comparing an integer value against a "BLOB" value. SQLite treats all integers and "REAL" values as smaller than a "BLOB" or text value. This means that for example $9 < '1'$ will be true, and may cause some issues if one is not careful with the affinity types.

One way to work around this is to make sure that creating tables based on some existing data is done entirely through SQLite commands, as SQLite uses the same affinity for a column as its source. If the table is made by first selecting some values through a select, keeping the values in memory in Python such as through variables, and then creating a table from these values, the affinities of the new table are not guaranteed to be the same as for the source.

7.8 Correctness

To ensure correctness, 49 tests were made using pytest [17]. These check if the results following a variety of operations, and using different queries for the views, are correct. The results of the views following a set of operations and a refresh should be the same as for a normal (non-materialized) view in a non-distributed setting, and the tests are used to verify that this is the case. Using the "pytest-cov" plugin the coverage was measured to be 89%.

7.9 Query design

The implementation makes frequent use of SQLite queries to manipulate the database. Queries in general can be written in a wide variety of ways and still yield the same result. How the query is written has a massive impact on how long it takes to execute, and when they are performed as regularly as in our refresh algorithms they are crucial in how the solution performs. The queries used in the current implementation have gone through multiple iterations, but may still be optimized further. As with any solution in a development process, the inner workings may change, and a query that was previously used may be replaced with a more suited one. By revisiting old queries, many performance improvements have been made. One example is the query used to identify and retrieve incoming tuples that have been updated. This query previously used four join statements to get tuples resulting from the incoming delta, also tracked in the CL table in addition to in the augmented view. The latter part, being included in the augmented view, is replaced with an additional select statement that checks if it exists, instead. This improves the refresh times drastically, going from a linear increase in time needed to refresh depending on the size of the data set, to a close to constant time needed.

7.10 Storage costs

Due to the relatively large amount of extra storage costs, one must consider if views are worth using. If the views are rarely used, or the disk usage is crucial for the device holding the views then the negative impact of having views on the site may outweigh the benefits. This is especially true for join views using incremental refresh as they use consume the most disk space seen in the experiments 6.2.3.

7.11 Future work

In the current implementation, there is no work done to ensure that one can create views from other views. This can be a useful mechanic and is something that SQLite allows for standard non-materialized views. As the current solution uses some of the inbuilt SynQLite methods to generate a delta file for a site, and the materialized views are made separately without initializing the tables as CRR-tables, it is not possible to create views from other views. One would need to create some functionality that allows for creating delta tables for views to be able to support this mechanic.

Another possible improvement would be to expand upon query support to allow for more possibilities such as aggregations, subqueries, outer joins, etc. This would improve the usefulness of these views and allow for a broader range of applicable use cases.

As previously discussed, using these views on a site may cause conflicts with any CRR tables on that same site. It would be useful if one could use both at the same site without conflicts. This can be done by making some new function for the CRR class, that allows for generating delta independently for views and CRR-tables and using separate time stamps for the two. This would allow for getting and applying changes separately, but issues may still arise as join-views and complete refresh will refresh the underlying base tables/CRR tables as well.

Additionally looking into using real-world network traffic for the system can also be done. The implementation should not be far from being able to handle these cases already, as everything is built with it in mind. One would have to test the system with multiple sites and replicas that reside on separate devices and ensure that the results remain correct and consistent.

7.12 Lessons learned

Through the work on this thesis, I have learned a lot about SQLite, its characteristics, and its quirks. Many behaviors have caused unintended results and have demanded some investigation to understand. Additionally, I have learned a lot about the importance of query design, as this can be crucial for the performance and the capabilities of the system. As the queries are executed frequently, it is essential to adjust them for their specific use case to reach acceptable results. Small changes can have huge impacts on the system.

I have also learned about exciting technologies such as CRRs and local-first

software in addition to the broad field of distributed systems. Debugging and testing are crucial aspects in these systems, as bugs and peculiarities may be hard to discover and get to the root of. Without the continuous use of the implemented test cases, ensuring that the results and behaviors of the solution are acceptable may have been insurmountable.

8

Conclusion

In this thesis, we have implemented and explored local-first relational views on top of SynQLite, an SQLite extension with CRR support. SynQLite keeps track of the history of changes and allows for generating a delta for base table changes. Previous works have implemented similar views using other technologies, but not in SQLite. This solution uses materialized views and supports both incremental and complete maintenance methods in order to apply changes made on other sites.

Applying only new changes to the view using the incremental refresh is done by utilizing causal lengths and provenance expressions. These allow us to keep track of the base tuples and different changes made to them and how they affect the end result that is the contents of the view. This thesis also presents an addition to the provenance expressions that allow for tracking update operations as well, and not only inserts and deletes.

Experiments are conducted on a simulated distributed setting and show a substantial improvement in refresh speed using the incremental approach in comparison to the complete refresh approach. The incremental refresh is close to constant in all three operations: insert, delete, and update, while the complete refresh is highly dependent on the size of the view in terms of the number of tuples. The complete approach does however perform better than the incremental approach in some cases, particularly where the number of operations are relatively large per refresh. One must therefore consider the use case and its characteristics, before deciding on what mode to use.

Bibliography

[1] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. 2nd. USA: Apress, 2010, pp. 59–59, 153–161. ISBN: 1430232250.

[2] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://www.agilemanifesto.org/.

[3] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)." In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7. ISBN: 1581131836. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502.

[4] Goretti K. Y. Chan, Qing Li, and Ling Feng. "Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study." In: *Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP*. DOLAP '99. Kansas City, Missouri, USA: Association for Computing Machinery, 1999, pp. 42–47. ISBN: 1581132204. DOI: 10.1145/319757.319787. URL: https://doi.org/10.1145/319757.319787.

[5] Surajit Chaudhuri and Moshe Y. Vardi. "Optimization of Real Conjunctive Queries." In: *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '93. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 59–70. ISBN: 0897915933. DOI: 10.1145/153850.153856. URL: https://doi.org/10.1145/153850.153856.

[6] Leonardo Weiss F. Chaves et al. "Towards Materialized View Selection for Distributed Databases." In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: Association for Computing Machinery, 2009, pp. 1088–1099. ISBN: 9781605584225. DOI: 10.1145/1516360.1516484. URL: https://doi.org/10.1145/1516360.1516484.

[7] Rada Chirkova and Jun Yang. "Materialized Views." In: *Foundations and Trends® in Databases* 4.4 (2012), pp. 295–405. ISSN: 1931-7883. DOI: 10.1561/1900000020. URL: http://dx.doi.org/10.1561/1900000020.

[8] D. E. Comer et al. "Computing as a Discipline." In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[9] Oracle Corporation. *Materialized View Concepts and Architecture*. URL: https://docs.oracle.com/cd/B10500_01/server.920/a96567/repmview.htm. (accessed: 26.05.2023).

[10] Lisa Crispin. "Driving Software Quality: How Test-Driven Development Impacts Software Quality." In: *IEEE Software* 23.6 (2006), pp. 70–71. DOI: 10.1109/MS.2006.157.

[11] Mihai Liviu Despa. "Comparative study on software development methodologies." In: *Database Systems Journal* (2014), pp. 37–56.

[12] Lars Marius Elvenes. "Materialized views in SQLite." Capstone Project, University of Tromsø. 2022.

[13] The PostgreSQL Global Development Group. *Materialized Views*. URL: https://www.postgresql.org/docs/current/rules-materializedviews.html. (accessed: 26.05.2023).

[14] Amarnath Gupta. "Data Provenance." In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 608–608. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1305. URL: https://doi.org/10.1007/978-0-387-39940-9_1305.

[15] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally." In: *SIGMOD Rec.* 22.2 (June 1993), pp. 157–166. ISSN: 0163-5808. DOI: 10.1145/170036.170066. URL: https://doi.org/10.1145/170036.170066.

[16] Martin Kleppmann et al. "Local-First Software: You Own Your Data, in Spite of the Cloud." In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://doi.org/10.1145/3359591.3359737.

[17] Holger Krekel et al. *pytest x.y*. 2004. URL: https://github.com/pytest-dev/pytest.

[18] Alexandros Labrinidis and Yannis Sismanis. "View Maintenance." In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 3326–3328. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_852. URL: https://doi.org/10.1007/978-0-387-39940-9_852.

[19] Mihai Letia, Nuno Preguiça, and Marc Shapiro. "Consistency without concurrency control in large, dynamic systems." In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 29–34. DOI: 10.1145/1773912.1773921.

[20] Microsoft. *Views*. URL: https://learn.microsoft.com/en-us/sql/relational-databases/views/views?view=azure-sqldw-latest. (accessed: 26.05.2023).

[21] Paulo Pintor, Rogério Luís de Carvalho Costa, and José Moreira. "Why- and How-Provenance in Distributed Environments." In: *Database and Expert Systems Applications*. Ed. by Christine Strauss et al. Cham: Springer International Publishing, 2022, pp. 103–115. ISBN: 978-3-031-12423-5.

[22] Owais Qayyum and Weihai Yu. "Toward Replicated and Asynchronous Data Streams for Edge-Cloud Applications." In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 339–346. ISBN: 9781450387132. DOI: 10.1145/3477314.3507687. URL: https://doi.org/10.1145/3477314.3507687.

[23] Abderrazak Sebaa and Abdelkamel Tari. "Materialized View Maintenance: Issues, Classification, and Open Challenges." In: *Int. J. Cooperative Inf. Syst.* 28 (2019), 1930001:1–1930001:59.

[24] SQLite. *Built-in Aggregate Functions*. URL: https://www.sqlite.org/lang_aggfunc.html. (accessed: 23.05.2023).

[25] SQLite. *CREATE INDEX*. URL: https://www.sqlite.org/lang_createindex.html. (accessed: 23.05.2023).

[26] SQLite. *Datatypes In SQLite*. URL: https://www.sqlite.org/datatype3.html. (accessed: 03.05.2023).

[27] SQLite. *INSERT*. URL: https://www.sqlite.org/lang_insert.html. (accessed: 23.05.2023).

[28] SQLite. *Most Widely Deployed and Used Database Engine*. URL: https://www.sqlite.org/mostdeployed.html. (accessed: 01.12.2022).

[29] SQLite. *SELECT*. URL: https://www.sqlite.org/lang_select.html. (accessed: 01.12.2022).

[30] SQLite. *SQLite File IO Specification*. URL: https://www2.sqlite.org/fileio.html. (accessed: 23.05.2023).

[31] SQLite. *The DBSTAT Virtual Table*. URL: https://www.sqlite.org/dbstat.html. (accessed: 12.0.5.2023).

[32] Michael Stonebraker. "Implementation of Integrity Constraints and Views by Query Modification." In: *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*. SIGMOD '75. San Jose, California: Association for Computing Machinery, 1975, pp. 65–78. ISBN: 9781450373289. DOI: 10.1145/500080.500091. URL: https://doi.org/10.1145/500080.500091.

[33] Iver Toft Tomter and Weihai Yu. "Augmenting SQLite for Local-First Software." In: *New Trends in Database and Information Systems*. Ed. by Ladjel Bellatreche et al. Cham: Springer International Publishing, 2021, pp. 247–257.

[34] Weihai Yu and Claudia-Lavinia Ignat. "Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge." In: *2020 IEEE International Conference on Smart Data Services (SMDS)*. 2020, pp. 113–121. DOI: 10.1109/SMDS49396.2020.00021.