Workload-based logical schema tuning

Broekman, Martijn S.

Award date:
2024

Link to publication

Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Database Research Group

Workload-based logical schema tuning

Master Thesis

M.S. Broekman

Supervisors:
prof. dr. George Fletcher
prof. dr. Johan Lukkien
dr. Odysseas Papapetrou

1.1

Eindhoven, March 2024

Abstract

In classical normalisation, the goal is to eliminate duplicate data in tables. This can only be obtained in Boyce-Codd Normal Form (BCNF). Lower normal forms increase duplicate data but have the advantage of having to join fewer tables, at the cost of having more expensive updates to maintain data consistency. In this work, we introduce two metrics: workload join efficiency $\ell_{\Theta}^{J,Total}$ and workload update inefficiency $\ell_{\Theta}^{U,Total}$. These two metrics evaluate the efficiency of a set of join and update actions Θ, by measuring the redundancy of attributes used in a join or update action. In particular, they inform us about the mean redundancy of the joined or updated attributes. The core concepts of increasing redundancy for attributes used in read actions and reducing redundancy for update actions are then incorporated into three variants of normalisation algorithms. In synthetic experiments and a case study, we find that these algorithms are able to create decompositions that have low redundancy for updated attributes and high redundancy for joined attributes, but that automatic schema generation can still create decompositions that hurt physical join performance due to the need for de-duplication when joining redundant attributes.

Preface

Completing this thesis has been a journey of ups and downs but, most importantly, personal growth. As I reflect on this journey, I am proud of what I have achieved and hope my findings are useful to the field of database design. I am very grateful for the support, guidance, and constructive feedback provided by my graduation supervisor, Professor George Fletcher. Our weekly meetings helped greatly in the completion of this project, guiding me when I got stuck and giving me focus on the important part of the project.

Contents

Contents **vii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Inspiration 1
- 1.3 Contributions 1

2 Practitioner's summery **3**

3 Preliminaries **6**
- 3.1 Functional Dependencies 6
- 3.2 Cardinality Constraints 7
- 3.3 Armstrong's Axioms 7
- 3.4 Attribute Closure 7
- 3.5 Lossless Decomposition 8
- 3.6 FD-preserving Decomposition 8
- 3.7 Classical normal forms 8

4 Workload-aware decomposition Assessment **11**
- 4.1 Introduction 11
- 4.2 Assessing efficiency 11
  - 4.2.1 Calculating join efficiency 11
  - 4.2.2 Update inefficiency 12
- 4.3 Workload-aware assessment 13

5 Workload-aware decomposition algorithms **16**
- 5.1 Introduction 16
- 5.2 Workload merging 16
  - 5.2.1 Create join tree 17
  - 5.2.2 Merge join trees 17
  - 5.2.3 Nodes selection 17
- 5.3 Workload aware algorithms 19
  - 5.3.1 2NF-FD Preserving algorithm 20
  - 5.3.2 HYBRID-FD Preserving algorithm 22
  - 5.3.3 HYBRID 2NF algorithm 22

6 General workload-aware decomposition performance **23**
6.1 Introduction 23
6.2 Motivation 23
6.3 Experimental design 23
6.4 Static analysis 25
6.5 Performance simulation 27
6.6 Discussion and findings 29

7 Micro-benchmark of generated decompositions **31**
7.1 Introduction 31
7.2 Workload 31
7.3 Decomposition 32
7.4 Discussion and findings 34

8 Future work **35**

9 Conclusions **36**

Bibliography **37**

Appendix **39**

A Workload Simulation **39**
A.1 Introduction 39
A.2 Motivation 39
A.3 Background 39
A.4 Schema Simulation 41
A.4.1 Modelling 41
A.4.2 Generation 41
A.5 Query Simulation 42
A.5.1 Modelling 42
A.5.2 Generation 42

List of Figures

5.1 Example join tree 1 18
5.2 Example join tree 2 18
5.3 Tree 1 merged into tree 2 19
5.4 Tree 2 merged into tree 1 19

6.1 Graph of the FDs in the china (left) and ncvoter (right) data sets 24
6.2 Visualization of the nodes visited with 128 queries with 4 joins each 24
6.3 Line plot of mean $\ell_{\Theta}^{U,total}$ for each update fractions for china data set 25
6.4 Line plot of mean $\ell_{\Theta}^{U,total}$ for each update fractions for ncvoter data set 25
6.5 Line plot of mean $\ell_{\Theta}^{J,total}$ for each update fractions for china data set 25
6.6 Line plot of mean $\ell_{\Theta}^{J,total}$ for each update fractions for ncvoter data set 25
6.7 Line plot of mean $\ell_{\Theta}^{U,total}$ for each # actions for china data set 26
6.8 Line plot of mean $\ell_{\Theta}^{U,total}$ for each # actions for ncvoter data set 26
6.9 Line plot of mean $\ell_{\Theta}^{J,total}$ for each # actions for china data set 26
6.10 Line plot of mean $\ell_{\Theta}^{J,total}$ for each # actions for ncvoter data set 26
6.11 Heat map of mean $\ell_{\Theta}^{J,total}$ for each # actions and update fraction for china data set 27
6.12 Heat map of mean $\ell_{\Theta}^{U,total}$ for each # actions and update fraction for china data set 27
6.13 Mean join query time on china data set 28
6.14 Box plot showing the median and interquartile range of the join query time 28
6.15 Box plot showing the median and interquartile range of the join query time split by update fraction 29
6.16 The mean number of tables tables per fraction 29

7.1 ncvoter decomposed using OPT 33
7.2 ncvoter decomposed using OPT 2NF max update = 1 33
7.3 ncvoter decomposed using OPT 2NF max update = 2 33
7.4 Section of the join tree for a query on the ncvoter FDs 33

A.1 TPC-C model [4] 40
A.2 Initial cardinality with $SF = 1$ 42
A.3 Initial cardinality with $SF = 2$ 42

List of Tables

2.1 Example table address information in the Netherlands 3
2.2 Example lossless decomposition of address information in the Netherlands 4

3.1 Table of to database theory notation and their definitions 6
3.2 Example table for a high school database 7
3.3 Example lossless decomposition of the high school database 8

4.1 Decomposition of Item and Sales 11
4.2 Updating and a price for the Item and Sales 12
4.3 fully normalised decomposition of Items, Sales, and Payments 14
4.4 Partially normalised decomposition of Items, Sales, and Payments 14

6.1 Difference # FDs original paper vs. mined 24

7.1 Manually selected FDs from ncvoter data set 31
7.2 Efficiency results of the different different decompositions 32

Chapter 1

Introduction

1.1 Motivation

In the last decade, the volume of data created has grown from 6.5 zettabytes in 2012, to 97 zettabytes in 2022. It is expected to grow to 160∼181 zettabytes in 2025 [13, 8]. Many modern solutions have been designed to address this explosive growth in data. For example, distributed computing or cloud computing, but when investigating the actual usage of database environments, the majority of developers use relational database management systems (RDBMS) [1]. Therefore, addressing data growth in relational databases is an important skill for a future software engineer.

An approach to addressing performance issues is through normalisation. Normalisation is the act of organising columns and tables of a database to reduce redundancy. The higher the level of normalisation, the less redundancy exists in a database. Less redundancy implies greater update performance, due to fewer duplicate values to update. However, the more redundancy is removed, the more joins need to be made to obtain the full database schema. Therefore, it is important to choose an appropriate level of normalisation, such that you can balance the performance of updates and joining tables. Higher levels of normalisation lead to better update efficiency, but worse join efficiency. However, in theory, lower normal forms, such as the second normal form (2NF) and the third normal form (3NF), are only seen as stepping stones to achieve the Boyce-Codd normal form (BCNF) [7]. The lower normal forms are not seen as goals of a design process themselves.

1.2 Inspiration

The core inspiration for this thesis is a recent work called "Logical Schema Design that Quantifies Update Inefficiency and Join Efficiency" by Link and Wei [10]. They discuss the limitations of classical normalisation in maintaining data consistency under updates and propose the usage of ℓ-Bounded Cardinality Normal Form (ℓ-BCNF) as a solution. This normal form requires that every value combination on the left-hand side of every non-trivial functional dependency does not occur in more than ℓ tuples. A decomposition in this normal form allows efficient updates and joins, and the level ℓ can be used to quantify incremental maintenance and join efficiency. The authors mention how their theory can be applied in practice. They mention that their theory should not blindly be applied, only when stable patterns of workloads have emerged. These patterns can tell you whether migrating to a new schema design will improve performance.

1.3 Contributions

In this work, we extend the definition of ℓ-BCNF to measure the efficiency of joins and updates in a decomposition for a given workload of joins and updates. By measuring the level ℓ only for the attributes that are used in the update or join queries, we can reason about the mean redundancy of the update queries and the reduced number of joins due to increased redundancy, separately. Using the new definitions of workload update inefficiency and workload join efficiency, we designed 3 normalisation algorithms. The algorithms allowed schemata to be created in 3NF or 2NF based on the workload of joins and updates. The goal is always to achieve a given level of redundancy for the updated attributes, while increasing the redundancy for the joined attributes.

Later in this work, we validated both the algorithms and the insight of the new metrics through two experimental studies. First, by creating a large number of different synthetic workloads to measure the general performance. Second, through a micro-benchmark of a small hypothetical workload. Here, we find that our algorithms can create decompositions with high redundancy for joined attributes and low redundancy for updated attributes. Furthermore, we are able to measure that higher redundancy can increase physical join performance in SQL but that having to remove duplicate values to join tables will hurt performance. The core of our findings are summarised for practitioners, so that they can use the theory in their database design process.

Chapter 2

Practitioner's summery

Designing a relational database schema can be difficult, especially when performance is taken into account. In this chapter, we want to give insight to practitioners into the balance of join performance vs. update performance, and why understanding how your database is used can help guide you in creating a better database design.

At the core of database schema design lies decomposition. This is the act of dividing a table into multiple tables with the goal of removing redundant duplicate values. However, a duplicate value is not always redundant, so how do we measure this? In database theory, there exists a concept of functional dependencies. A functional dependency represents a one-way relationship between a set of columns A and another set of columns B, where for each unique value in the columns A, there is always only one value associated with the value in the columns B. We can denote these relationships as $A \rightarrow B$. Redundancy can occur when multiple functional dependencies are in the same table. For example, in the Netherlands, every address consists of: (street, number, postcode, city). for these columns, the following functional dependencies hold:

- $\{street, number, city\} \rightarrow \{postcode\}$
- $\{postcode\} \rightarrow \{street, city\}$

Table 2.1 shows an example of a table containing the address information. We can observe that for a single postcode, the street and the city are always the same, but there are 25 rows with the same postcode of 5612AZ. All these rows also have the same city and street. The maximum redundancy of 25 for the postcode is called the Cardinality Constraint. We can remove this duplication by decomposing the table into (Street, City, Postcode) and (Number, Postcode) as seen in Table 2.2. Is this new decomposition of address information better? To answer this question, we will give reasons for and against decomposition.

The first advantage of decomposing is that less storage is needed. Due to the postcode becoming a key in the new table, the Cardinality Constraint of 25 is reduced to no duplication, thus only 1

Street	Number	City	Postcode
Kruisstraat	1	Eindhoven	5612HK
Den Dolech	1	Eindhoven	5612AZ
...	...	...	...
Den Dolech	25	Eindhoven	5612AZ
Den Dolech	26	Eindhoven	5612BA
Heidelberglaan	8	Utrecht	3584CS
Leidsestraat	1	Amsterdam	7834DS
...	...	...	...

Table 2.1: Example table address information in the Netherlands

Street	City	Postcode
Kruisstraat	Eindhoven	5612HK
Den Dolech	Eindhoven	5612AZ
Den Dolech	Eindhoven	5612BA
Heidelberglaan	Utrecht	3584CS
Leidsestraat	Amsterdam	7834DS
...	...	...

Number	Postcode
1	5612HK
1	5612AZ
...	...
25	5612AZ
26	5612BA
8	3584CS
1	7834DS
...	...

Table 2.2: Example lossless decomposition of address information in the Netherlands

occurrence for each postcode. Furthermore, it is less prone to errors in queries. If we want to know how many streets there are for a postcode, we can do a GROUP BY statement on the postcode and COUNT the city. In the original schema, we would have to eliminate duplicate values using a DISTINCT. This also means that this query is more expensive since more rows need to be scanned to answer the same query.

The downside of decomposing is that if we want full address information, we need to join the tables back together on the postcode. The Cardinality Constraint also informs us on the maximum number of joins to be made, in this case 25 instances of postcode. Joining tables decreases performance over selecting all values from a single table. Furthermore, the columns of functional dependency $\{street, number, city\} \rightarrow \{postcode\}$ no longer exist in a single table. In data-base theory, this means that this functional dependency is "lost". This implies that in case there is a need to update a column such as a house number, then we would first join the tables to validate if it is allowed. For example, if the house number of postcode 5612BA is changed to 25 there would be a conflict, since the address Den Dolech 25 would have two postcodes. However, functional dependencies are not always lost when decomposing tables. If we introduce buildings owned by corporations into this example, then the functional dependencies would become:

- $\{street, number, city\} \rightarrow \{postcode, building\}$
- $\{postcode\} \rightarrow \{street, city\}$
- $\{building\} \rightarrow \{corporation\}$

In this case $\{building\} \rightarrow \{corporation\}$ is not lost if (building, corporation) would be its own table and the address information only stores the building. It would only increase update performance when a building changes corporation, since only one row needs to be updated. However, the joining cost still exists.

To help make decisions about whether or not to decompose tables, you can sum the cardinality constraints of all functional dependencies. A higher score indicates better join performance, and a lower score indicates faster updates. However, not all columns are joined and updated equally. Therefore, we introduced two metrics that look at the usage of columns. The first metric is the mean Cardinality Constraints of all updated columns. For example, if only the field "number" was updated, then it would have no impact on the score if the tables were decomposed or not, since in both cases the duplication of the column does not change. The second metric is the mean Cardinality Constraints of all the read columns. When decomposing, you want to keep the metric for the updated columns low, since an higher score means more duplicate values to update. For the joined columns, you want to increase the score, since having more redundant values means less joins to make. You can see the full definition of these metrics in chapter 4.

From our experiments with automatically generating decompositions using these metrics, we can give the following recommendations. In general, try to decompose as much as possible but do not lose any functional dependencies. For each possible lost functional dependency, decompose only if the update performance loss is not an issue and the tables do not have to be joined often. In chapter 6, we show that you can safely lose functional dependencies without hurting performance, if they are not used.

On the other hand, if two tables in a fully decomposed schema are always joined, you can choose to merge them back together. However, understand that duplicate elimination is an expensive operation. Therefore, be sure that there are no queries that use duplicate data separately and that updating duplicated data will be more expensive. In chapter 6, we evaluated our decomposition that did not check this and found that having to use GROUP BY to join duplicate values significantly affects performance.

Chapter 3

Preliminaries

In these preliminaries, we describe multiple concepts related to database theory, and in Table 3.1 we have some notation that we will use in this work.

Notation	*Definition*
relation schema R	finite set R of attributes A, each having a domain $dom(A)$ comprising potential values.
database schema $\mathcal{D}$	finite set $\mathcal{D} = \{R_1, \ldots, R_n\}$ for relational schemata
tuple t over R	function $t : R \to \bigcup_{A \in R} dom(A)$ with $t(A) \in dom(A)$
relation r over R	finite set r of tuples over R
FD $X \to Y$	for all tuples $t, t' \in r$, if $t(X) = t'(X)$, then $t(Y) = t'(Y)$
Non-trivial FD $X \to Y$	An FD is not-trivial if $Y \nsubseteq X$
Cardinality Constraint (CC) $card(X) \leq \ell$	For a positive integer ℓ, the CC $card(X) \leq \ell$ says that every instance can have up to ℓ different records with matching values on all attributes in X.
Set Σ	Set containing all FDs and CCs
semantic closure Σ^*	All FDs implied by Σ
Projection $\Sigma[X]$	Subset of all FDs/CCs $A \to B \in \Sigma^*$ for which $AB \subseteq X$ holds

Table 3.1: Table of to database theory notation and their definitions

3.1 Functional Dependencies

We denote table R in the database schema $\mathcal{D}$ as a set of attributes. If we then have the Functional Dependency (FD) $X \to Y$ where $XY \subseteq R$ this implies that for every unique value of A, there is a unique corresponding value of B. In other words, if you know the value of A, you can uniquely determine the value of B [11].

For example, if we have a database for a small high school, where each subject is given by a single instructor and an instructor can only give one subject in a time slot, then we could have the following set of FDs $\Sigma = \{(Instructor, TimeSlot \to Subject), (Subject \to Instructor)\}$. Table 3.2 shows the rows in this database. It can be seen that for the combination of *Instructor* and *TimeSlot* there is always a distinct subject. Furthermore, does each *Subject* have a unique *Instructor*, chemistry is always given by Peter, and math and physics are always given by John.

Instructor	TimeSlot	Subject
John	1	math
John	2	math
John	3	physics
Peter	1	chemistry
Peter	...	chemistry
Peter	8	chemistry

Table 3.2: Example table for a high school database

3.2 Cardinality Constraints

The set of positive integers, including infinity, is denoted as $\mathbb{N}^{\infty}_{\geq 1}$. A cardinality constraint (CC) in a relation schema R is an expression of the form $card(X) \leq \ell$, where $X \subseteq R$ and $\ell \in \mathbb{N}^{\infty}_{\geq 1}$. This means that a relation r over R satisfies CC if there are no more than ℓ different tuples in r that have the same values for all attributes in X.

For our high school example of Table 3.2, the CCs are $\{card(Instructor, TimeSlot) \leq 1, card(Subject) \leq 8\}$. If a new teacher would teach chemistry instead of Peter, then we would have to update all 8 occurrences. If only one row were changed, then there would be two different people give chemistry. This means that FD $Subject \rightarrow Instructor$ would no longer hold.

3.3 Armstrong's Axioms

Armstrong's axioms is a set of 3 inference rules used for the logical implication of FDs [2]. he set consists of the following rules:

- Reflexivity: $\frac{}{XY \rightarrow Y}$
- Extension: $\frac{X \rightarrow Y}{X \rightarrow XY}$
- Transitivity: $\frac{X \rightarrow Y \quad Y \rightarrow Z}{X \rightarrow Z}$

From a set of FD Σ, Armstrong's axioms can be applied to derive further FDs from the ones in Σ. An FD $X \rightarrow Y \notin \Sigma$ is considered implied by Σ, if Y can be obtained from X by applying the inference rules on X.

To illustrate how new FDs can be inferred using Armstrong's axioms, we will add the FD $Student, TimeSlot \rightarrow Subject$ to the FDs of the high school database Σ. This FD represents that a student can only have one subject in a given time slot. If we would like to know which instructors a student has in his different time slots, then we can apply the transitivity rule on $Student, TimeSlot \rightarrow Subject$ and $Subject \rightarrow Instructors$ to obtain $Student, TimeSlot \rightarrow Instructor$.

3.4 Attribute Closure

For the relation schema R, let A^+ with $A \subseteq R$ represent the attribute closure under the set of FDs Σ. Attribute closure A^+ is defined as the maximal subset $B \subseteq R$ such that $A \rightarrow B$ is implied by the FDs in Σ [7].

For example, compute the closure of $Student, TimeSlot$ in the high school database then we can obtain $(Student, TimeSlot)^+$ in the following steps:

1. $(Student, TimeSlot)^+ = Student, TimeSlot$
   reflectivity on $Student, TimeSlot$
2. $(Student, TimeSlot)^+ = Student, TimeSlot, Subject$
   extension with FD $Student, TimeSlot \rightarrow Subject$

3. $(Student, TimeSlot)^{+} = Student, TimeSlot, Subject, Instructor$
   transitivity with FD $Subject \rightarrow Instructor$

3.5 Lossless Decomposition

Decomposition is the act of dividing a relational schema R into a finite set $\mathcal{D} = \{R_1, \dots, R_n\}$ for relational schemata. This process is considered lossless if any instance r of R it holds that $r = \pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_n}(r)$. For example, divide the attributes $Instructor, TimeSlot, Subject$ of Table 3.2 into $Instructor, Subject$ and $TimeSlot, Subject$ as seen in Table 3.3. This decomposition is then considered lossless since the original table can be re-obtained by joining both on the $Subject$ attribute.

Instructor	Subject
John	math
John	physics
Peter	chemistry

TimeSlot	Subject
1	math
2	math
3	physics
1	chemistry
...	chemistry
8	chemistry

Table 3.3: Example lossless decomposition of the high school database

3.6 FD-preserving Decomposition

A decomposition $\mathcal{D}$ of (R, Σ) is FD-preserving, if all dependencies in Σ are part of the dependencies projections $\Sigma[S]$ of all tables $S \in \mathcal{D}$, or can be derived from $\bigcup_{S \in \mathcal{D}} \Sigma[S]$ using Armstrong's axioms. For our running example, this does not hold for the decomposition created in Table 3.3, since the FD $Instructor, TimeSlot \rightarrow Subject$ is not in any of the tables projection, nor can it be derived.

3.7 Classical normal forms

The goal of classical normalisation is to reduce redundant values in database tables by eliminating duplicate values through decomposition. Four normal forms are important for this work. We will give the definitions for the first, second, and third normal forms and Boyce-Codd normal form (BCNF) as defined in the literature [7, 11].

In order to understand higher-levels of normal forms, we the following definitions of superkey and subkey:

- **Candidate key**: We say that $X \subseteq R$ is a candidate key for R if and only if it holds that for each instance r of R the value of $X(r)$ is unique and there is no subset $A \subseteq X$ where $A(r)$ is unique.
- **Superkey**: We say that $X \subseteq R$ is a superkey for R if and only if any valid instance r of R, and any two tuples $t, t' \in r$, it is the case that if $X(t) = X(t)$ then for all attributes $Y \in R$, $Y(t) = Y(t')$ should hold.
- **Subkey**: We say that $X \subseteq R$ is a subkey of R if and only if each attribute in X is contained in some candidate key.

Using these definitions, we describe the various normal forms. Each subsequent form always encompasses the lower normal form, but not vice versa. The definitions are as follows:

- **First normal form (1NF)**: Let the relation R have attributes $A_1, \ldots, A_n$, of primitive types $T_1, \ldots, T_n$, respectively. Then R is in the first normal form (1NF) if and only if, for all tuples t appearing in R, the value of the attribute A_i in t is of type T_i $(i = 1, \ldots, n)$.
- **Second normal form (2NF)**: A decomposition $\mathcal{D}$ of (R, Σ) is in 2NF if and only if, for every non-trivial FD $X \to Y \in \Sigma$, at least one of the following statements is true:
  1. X is a superkey
  2. Y is a subkey
  3. X is not a subkey
- **Third normal form (3NF)**: A decomposition $\mathcal{D}$ of (R, Σ) is in 3NF if and only if, for every non-trivial FD $X \to Y \in \Sigma$, at least one of the following statements is true:
  1. X is a superkey
  2. Y is a subkey
- **Boyce-Codd normal form (BCNF)**: A decomposition $\mathcal{D}$ of (R, Σ) is in BCNF if and only if, for every non-trivial FD $X \to Y \in \Sigma$, X is a superkey.

For our Table 3.2 with attributes $Instructor, TimeSlot, Subject$ is in 3NF. FD $Instructor, TimeSlot \to Subject$ statement 1 of 3NF is valid, since the LHS is a superkey. For FD $Subject \to Instructor$ statement 2 holds because the *instructor* is a subkey.

Chapter 4

Workload-aware decomposition Assessment

4.1 Introduction

Link et al. [10] established the family of ℓ-Bounded Cardinality Normal Forms (ℓ-BCNF). These normal forms are a relaxed version of BCNF. Where in BCNF for all nontrivial FD $X \rightarrow Y$, $card(X) \leq 1$ holds, and X is a superkey. In ℓ-BCNF they relax this definition to where $card(X) \leq \ell$ instead of $card(X) \leq 1$. In other words, they allow a Left Hand Side (LHS) of an FD to have at most ℓ occurrences. They use this level ℓ to assess the join and update efficiency of a decomposition. In this chapter, we are going to describe how this level ℓ is defined in the original work and we use it to assess the join and update efficiency of a given workload.

4.2 Assessing efficiency

4.2.1 Calculating join efficiency

The ℓ-join efficiency is defined as the maximum number of tuples ℓ that have matching values in the LHS of a nontrivial FD. In classical normalisation such as BCNF the goal is to obtain an LHS with no matching values on the LHS, thus $\ell = 1$. Consequently, having a higher level ℓ than 1 means that further normalisation would be possible. However, it also means that fewer tables would have to join, since the schema is not fully normalised. As a consequence, is the performance better in retrieving all tuples from a decomposition with a greater level of ℓ, at the cost of more storage due to the number of duplicate values.

Example 4.1: the ℓ-join efficiency of the schema ($R_1 = \{SaleID, ItemID, Name, Price\}, \Sigma = \{\{ItemID \rightarrow Name, Price\}, \{SaleID \rightarrow ItemID\}, card(ItemID) \leq 1000\}$) is 1000, since the RHS $ItemID$ has a maximum of 1000 redundant values in R_1. However, if we decompose R_1 into $R_2 = \{ItemID, Name, Price\}$ and $R_3 = \{SaleId, ItemID\}$ as seen in Table 4.1, then the ℓ-join efficiency would be 1 since $ItemId$ no longer has redundant values.

SaleID	ItemID	Name	Price
s_1	1	Shirt	20
...	...	...	...
s_{1000}	1	Shirt	20
s_{1001}	2	Shoe	40

$=$

ItemID	Name	Price
1	Shirt	20
2	Shoe	40

$\bowtie$

SaleID	ItemID
s_1	1
...	...
s_{1000}	1
s_{1001}	2

Table 4.1: Decomposition of Item and Sales

To calculate the ℓ-join efficiency of a decomposition $\mathcal{D}$ Link et al. introduced the *join-supportive attribute subsets* $JS_{\mathcal{D}}^{R,\Sigma}$[10]. Equation 4.1 shows how the subset is defined. This set is a union of two subsets. The first subset consists of tables $S \in \mathcal{D}$ that contain an attribute X_k, where this attribute functionally determines S through all FDs in Σ. The second subset is similar to the first one, but it contains the keys X in table S that do not determine S itself.

$$\begin{aligned} JS_{D}^{R,\Sigma} =& \{S : X_k | \exists S \in \mathcal{D} \exists X_k \to Y \in \Sigma[S], \Sigma \vDash X_k \to S\} \cup \\ & \{S : X | \exists S \in \mathcal{D} \exists X \to Y \in \Sigma[S], \Sigma \nvDash X \to S\} \end{aligned} \tag{4.1}$$

The level of ℓ-join efficiency of $\mathcal{D}$ is defined in Equation 4.2, where $\ell_X := \min\{\ell \mid \Sigma \vDash card(X) \leq \ell\}$. All redundant LHS $S : X \in JS_{D}^{R,\Sigma}$ have the value of ℓ_X and the superkeys $S : X_k \in JS_{D}^{R,\Sigma}$ have a value of 1. Of these values, the supremum is the ℓ-join efficiency.

To illustrate how this formula works in practice, we compute the ℓ-join efficiency for the decompositions of Example 4.1. The *join-supportive attribute subsets* $JS_{\mathcal{D}}^{R,\Sigma}$ for R_1 is $JS_{\mathcal{D}}^{R,\Sigma} = \{R_1 : \{X_k : SaleId, X : ItemId\}\}$. $SaleId$ is the superkey of R_1 therefore it has a redundancy of 1, and $ItemId$ has a redundancy of $card(ItemID) \leq 1000$ therefore it contributes 1000. Thus, is the ℓ-join efficiency of this decomposition 1000. For the decomposition $\mathcal{D} = \{R_2, R_3\}$, the *join-supportive attribute subsets* transforms into $JS_{\mathcal{D}}^{R,\Sigma} = \{R_2 : \{X_k : ItemID\}, R_3 : \{X_k : SaleID\}\}$. This results in a ℓ-join efficiency of 1 since both RHS attributes are now superkeys of their respective tables.

$$\ell_{\mathcal{D}}^{J} = \sup\{\ell_X | S : X \in JS_{D}^{R,\Sigma}\} \cup \{1 | S : X_K \in JS_{D}^{R,\Sigma}\} \tag{4.2}$$

In addition to the notion of join efficiency $\ell_{\mathcal{D}}^{J}$, the authors also introduced the total join efficiency $\ell_{\mathcal{D}}^{J,total}$. Instead of the maximum level of join efficiency, this is a sum over the join efficiency of all tables $S \in \mathcal{D}$.

$$\ell_{\mathcal{D}}^{J,total} = \sum_{S:X \in JS_{D}^{R,\Sigma}} \ell_X + \sum_{S:X_K \in JS_{D}^{R,\Sigma}} 1 \tag{4.3}$$

4.2.2 Update inefficiency

The counterpart of ℓ-join efficiency is the ℓ-update inefficiency. The same level ℓ also tells us the maximum number of values that need to be updated to maintain consistency. If a tuple $t \in r$ in the domain of attribute Y for the FD $X \to Y$ is changed, then all other tuples $t' \in r$ with $t(X) = t'(X)$ also need to be updated, otherwise $t(Y) = t'(Y)$ would not hold.

Example 4.2: If we update *Price* for a single *SaleID* in the schema $(R_1 = \{SaleID, ItemID, Name, Price\}, \Sigma = \{\{ItemID \to Name, Price\}, \{SaleID \to ItemID\}, card(ItemID) \leq 1000\})$ would break the FD $ItemID \to Name, Price$, since there are now up to 999 occurrences that have a different *Price* for the same *ItemID* as the updated row. Therefore, in order to maintain the FD we need to update these 999 occurrences to the same *Price*. In Table 4.2 visualise this problem.

SaleID	ItemID	Name	Price		SaleID	ItemID	Name	Price
s_1	**1**	Shirt	**25**		s_1	**1**	Shirt	**25**
...	...	...	...	$\to$	...	...	...	...
s_{1000}	1	Shirt	20		s_{1000}	1	Shirt	25
s_{1001}	2	Shoe	40		s_{1001}	2	Shoe	40

Table 4.2: Updating and a price for the Item and Sales

The calculation of the ℓ-update inefficiency is similar to the update efficiency. We create the *update-critical attribute subsets* by combining the *join-supportive attribute subsets* with any lost FD due to the decomposition. Equation 4.4 shows the formal definition of the subset.

$$UC_D^{R,\Sigma} = JS_D^{R,\Sigma} \cup \{X \subseteq R | \exists X \to Y \in (\Sigma - (\bigcup_{S \in D} \Sigma[S])^+)\} \tag{4.4}$$

The level of update inefficiency and total level of update inefficiency are similar to their join efficiency counterpart. The update inefficiency $\ell_{\mathcal{D}}^U$ is the supremum of all levels ℓ_X and the total update inefficiency is the sum of all levels ℓ_X.

$$\ell_{\mathcal{D}}^J = \sup\{\ell_X | S : X \in UC_D^{R,\Sigma}\} \cup \{1 | S : X_K \in UC_D^{R,\Sigma}\} \tag{4.5}$$

$$\ell_{\mathcal{D}}^{U,total} = \sum_{S:X \in UC_D^{R,\Sigma}} \ell_X + \sum_{S:X_K \in UC_D^{R,\Sigma}} 1 \tag{4.6}$$

4.3 Workload-aware assessment

In this section, we describe our contributions to the assessment of update and join efficiency by making the metrics workload-aware. The current problem with the definitions of ℓ-join efficiency and ℓ-update inefficiency of Link et al. [10] is that they are not bounded by the usage of FDs in Σ. To solve this, we define the set of actions Θ that consists of a tuple $(Q_R, Q_U) \in \Theta$ where $Q_R, Q_U \subseteq R$. Q_R represents all attributes in R that are read for a given action, and Q_U represent the attributes that are updated.

Instead of evaluating a decomposition using $\ell_{\mathcal{D}}^{J,total}$ and $\ell_{\mathcal{D}}^{U,total}$, we evaluated the join efficiency and the update inefficiency with respect to the set Θ. First, we express the join efficiency in terms of a read action $Q_R \in \Theta$. Its join efficiency of is similar to that of the entire decomposition, but instead it being the sum of all cardinalities of $S : X, X_k \in JS_D^{R,\Sigma}$ is it the sum of only the attributes that are used to create Q_R. To calculate the score for Q_R, we filter the *join-supportive attribute subsets* $JS_{\mathcal{D}}^{(R,\Sigma)}$, by only selecting the $X, X_K \in JS_{\mathcal{D}}^{(R,\Sigma)}$ which are keys of the subsets of Q_R. We denote this new subset as $JS_{Q_R}^{(R,\Sigma)}$, and for each value in the subset we calculate the total level $\ell_{Q_R}^{J,Total}$ by adding the cardinality of each key.

$$\begin{aligned} JS_{Q_R}^{(R,\Sigma)} :=& \{S : X_k \in JS_D^{R,\Sigma} \mid \exists Y \subseteq Q_R, \Sigma \models X_k \to Y\} \cup \\ & \{S : X \in JS_D^{R,\Sigma} \mid \exists Y \subseteq Q_R, \Sigma \models X \to Y\} \end{aligned} \tag{4.7}$$

$$\ell_{Q_R}^{J,Total} = \sum_{S:X \in JS_{Q_R}^{(R,\Sigma)}} \ell_X + \sum_{S:X_K \in JS_{Q_R}^{(R,\Sigma)}} 1 \tag{4.8}$$

Theorem 1 For every read action $Q_R \in \Theta$ with $Q_R \subseteq R$ its total score $0 \le \ell_{Q_R}^{J,Total} \le \ell_{\mathcal{D}}^{J,Total}$

Proof To prove the upper bound $\ell_{Q_R}^{J,Total} \le \ell_{\mathcal{D}}^{J,Total}$, we use the following argument. $JS_{Q_R}^{(R,\Sigma)}$ and $JS_{\mathcal{D}}^{(R,\Sigma)}$ are both sets such that $JS_{Q_R}^{(R,\Sigma)} \subseteq JS_{\mathcal{D}}^{(R,\Sigma)}$, since $JS_{Q_R}^{(R,\Sigma)}$ is created by filtering $JS_{\mathcal{D}}^{(R,\Sigma)}$. $\ell_{Q_R}^{J,Total}$ and $\ell_{\mathcal{D}}^{J,Total}$ are summations of all X and X_K in $JS_{Q_R}^{(R,\Sigma)}$ and $JS_{\mathcal{D}}^{(R,\Sigma)}$. Since every element in $JS_{Q_R}^{(R,\Sigma)}$ is also in $JS_{\mathcal{D}}^{(R,\Sigma)}$ (because $JS_{Q_R}^{(R,\Sigma)} \subseteq JS_{\mathcal{D}}^{(R,\Sigma)}$), you can break down the sum over $JS_{\mathcal{D}}^{(R,\Sigma)}$ into two parts; the sum over the common elements in both sets, and the sum over the elements in $JS_{\mathcal{D}}^{(R,\Sigma)}$ but not in $JS_{Q_R}^{(R,\Sigma)}$.

$$\ell_{\mathcal{D}}^{J,Total} = \ell_{Q_R}^{J,Total} + (\sum_{S:X \in (JS_{\mathcal{D}}^{(R,\Sigma)} \setminus JS_{Q_R}^{(R,\Sigma)})} \ell_X + \sum_{S:X_K \in (JS_{\mathcal{D}}^{(R,\Sigma)} \setminus JS_{Q_R}^{(R,\Sigma)})} 1) \tag{4.9}$$

The summations over $(JS_{\mathcal{D}}^{(R,\Sigma)} \setminus JS_{Q_R}^{(R,\Sigma)})$ represent the elements in $JS_{\mathcal{D}}^{(R,\Sigma)}$ but not in $JS_{Q_R}^{(R,\Sigma)}$. Since $JS_{Q_R}^{(R,\Sigma)}$ is a subset of $JS_{\mathcal{D}}^{(R,\Sigma)}$, we can conclude that:

$$\sum_{S:X\in(JS_{\mathcal{D}}^{(R,\Sigma)}\setminus JS_{Q_R}^{(R,\Sigma)})} \ell_X + \sum_{S:X_K\in(JS_{\mathcal{D}}^{(R,\Sigma)}\setminus JS_{Q_R}^{(R,\Sigma)})} 1 \geq 0 \tag{4.10}$$

Therefore,

$$\ell_{Q_R}^{J,Total} \leq \ell_{\mathcal{D}}^{J,Total} \tag{4.11}$$

The lower bound of $0 \leq \ell_{Q_R}^{J,Total}$ is trivially proven by the fact if $JS_{Q_R}^{(R,\Sigma)} = \emptyset$ then the summations over these sets will yield 0.

Example 4.3: To illustrate how $\ell_{Q_R}^{J,Total}$ is computed, we have the following set of FDs/CCs:

- $ItemID \rightarrow Name, Price$; $card(ItemID) \leq 1$
- $SaleID \rightarrow ItemID$; $card(SaleID) \leq 1000$
- $PaymentID \rightarrow Amount, SaleId$; $card(PaymentID) \leq 10$

If then have a fully and a partially normalised schema in Table 4.3 and Table 4.4 respectively. For the read action $Q_R = \{PaymentID, Amount, SaleID, ItemID\}$, we calculate $\ell_{Q_r}^{J,total}$ for the fully normalised decomposition, which does not contain redundant FDs. We only read the items, and payments, therefore, is $\ell_{Q_R}^{J,total} = 2$. If we calculate the cost of the same read action with the partially denormalised schemata in Table 4.4, then $\ell_{Q_R}^{J,total}$ is also 2 since the redundant sale is not read. On the contrary, for the total join efficiency $\ell_{\mathcal{D}}^{J,Total}$ the redundant sale in partially normalised decomposition would increase the join efficiency by 1000, even though these attributes are not used.

ItemID	Name	Price
1	Shirt	20
2	Shoe	40

SaleID	ItemID
s_1	1
...	...
s_{1000}	1
s_{1001}	2

PaymentID	Amount	SaleID
p_1	1	s_1
...	...	
p_{10}	5	s_1
p_{11}	2	s_2

Table 4.3: fully normalised decomposition of Items, Sales, and Payments

SaleID	ItemID	Name	Price
s_1	1	Shirt	20
...	...	...	...
s_{1000}	1	Shirt	20
s_{1001}	2	Shoe	40

PaymentID	Amount	SaleID
p_1	1	s_1
...	...	
p_{10}	5	s_1
p_{11}	2	s_2

Table 4.4: Partially normalised decomposition of Items, Sales, and Payments

Update inefficiency can also be defined in terms of an action $(Q_R, Q_U) \in \Theta$. To find the cardinality of the updated tuples, we filter the *update-critical attribute subsets* [10], by selecting all the keys that determine the updated tuples. Using this filtered set $UC_{Q_U}^{(R,\Sigma)}$, we sum all levels of update inefficiency to find the total update inefficiency $\ell_{Q_U}^{U,total}$ of the update Q_U.

$$\begin{aligned} UC_{Q_U}^{(R,\Sigma)} :=& \{S : X_K \in UC_D^{R,\Sigma} \mid \exists Y \subseteq Q_U, \Sigma \vDash X_K \to Y\} \cup \\ & \{S : X \in UC_D^{R,\Sigma} \mid \exists Y \subseteq Q_U, \Sigma \vDash X \to Y\} \end{aligned} \tag{4.12}$$

$$\ell_{Q_U}^{U,total} = \sum_{X \in UC_{Q_U}^{(R,\Sigma)}} \ell_X + \sum_{X_k \in UC_{Q_U}^{(R,\Sigma)}} 1 \tag{4.13}$$

Theorem 2 For every update action $Q_U \in A$ with $Q_U \subseteq R$ its total score $0 \leq \ell_{Q_U}^{U,Total} \leq \ell_{\mathcal{D}}^{U,Total}$
Proof The proof technique employed in Theorem 1 can be extended to prove Theorem 2. Instead of the *Join supportive attribute subsets* $JS_{\mathcal{D}}^{(R,\Sigma)}$ and $JS_{Q_R}^{(R,\Sigma)}$ we will apply the same proof to the *update critical attribute subsets* $UC_{\mathcal{D}}^{(R,\Sigma)}$ and $UC_{Q_U}^{(R,\Sigma)}$.

Example 4.4: To illustrate how $\ell_{Q_U}^{U,total}$ is calculated, we will show an example of the decompositions of tables 4.3 and 4.4. The action $(\{SaleID, ItemID, Price\}, Price)$ produces $UC_{Price}^{(R,\Sigma)} = \{Items : \{X_k : ItemID\}\}$ for the fully normalised decomposition, and $UC_{Price}^{(R,\Sigma)} = \{Sales : \{X : ItemID\}\}$ for the partial normalised decomposition. Therefore is $\ell_{Price}^{U,total} = 1$ for the fully normalised decomposition and $\ell_{Price}^{U,total} = 1000$ for the partial normalised decomposition.

Finally, we create the definitions of the total join efficiency and update inefficiency of Θ as mean levels of the read and update actions:

$$\ell_{\Theta}^{J,total} = \frac{\sum_{Q_R \in \Theta} \ell_{Q_R}^{J,total}}{|\Theta|} \tag{4.14}$$

$$\ell_{\Theta}^{U,total} = \frac{\sum_{Q_U \in \Theta} \ell_{Q_U}^{U,total}}{|\Theta|} \tag{4.15}$$

Chapter 5

Workload-aware decomposition algorithms

5.1 Introduction

In this chapter, we will describe algorithms to normalise schemata for a given workload. These algorithms will transform a set of FDs/CCs Σ into a decomposition that maximises $\ell_{\Theta}^{J,Total}$ and minimises $\ell_{\Theta}^{U,Total}$. To achieve this, we need to simplify the read and write actions of the workload Θ into two sets. The first set denoted *Single* contains FDs $X \rightarrow Y \in \Sigma$ that should have low redundancy. The second set denoted *Joined* contains multiple subsets of FDs in Σ that are often joined. We create three workload-aware variants of the normalisation algorithms described by Link et al. [10].

5.2 Workload merging

To create a decomposition that is optimised for a given workload, we need to know how each read or update action affects FDs in Σ. If two FDs are often read together, then a decomposition should have these two FDs in a single table. However, if an FD is updated often, then the redundancy of that FD should be minimised by normalisation. Since there can be a lot of overlap between actions, we can simplify the workload by merging these similarities into the sets *Single* and *Joined*.

To describe how sets *Single* and *Joined* are created, we first formalise the problem. Given a set of FDs/CCs Σ and a workload Θ consisting of read and update actions $(Q_R, Q_U) \in \Theta$, we need to divide the FDs into subsets of Σ so that each FD $X \rightarrow Y \in \Sigma$ exists only in one of these subsets. We denote this set of subsets as T. Using T we can create *Single* and *Joined* as $Single = \{X \in T \mid |X| = 1\}$ and $Joined = \{X \in T \mid |X| > 1\}$. T is created in the following steps:

1. Transform the read actions into a join tree
2. Merge join trees
3. Select nodes from the join trees

5.2.1 Create join tree

The first step in creating sets $Single$ and $Joined$ is to transform the FDs in the projection $\Sigma[Q_R]$ for all read actions $Q_R \in \Theta$ into a data structure that allows one to compare the FDs that overlap between different read actions. We choose to use the tree data structure with the leafs representing the FDs and the nodes the "joins". We will denote this data structure as a "join tree".

A read action $Q_R \in \Theta$ is transformed into a join tree using its FDs $\Sigma[Q_R]$. Two FDs $X \rightarrow A, Y \rightarrow B \in \Sigma[Q_R]$ are considered "joined" if $Y \subseteq XA$, that is, if a left FD contains all attributes of the right FD LHS. All the attributes from the LHS are needed since joining on a partial LHS will cause data duplication. All FDs in a read action are transformed into a list of joined FD pairs. This list is then ordered from low to high on the basis of the cardinality of the right FD's LHS. For our example of $X \rightarrow A, Y \rightarrow B \in \Sigma[Q_R]$ the ordering is based on $CARD(Y) \leq \ell$. The pairs are ordered for two reasons. First, it causes consistency when generating a tree from a read action. Second, joining from low-cardinality FDs first is preferred, since gradual increase of cardinality causes less redundancy if not all FDs in the tree are joined.

From the ordered list of joined FDs, we create a join tree. For each FD in a read action's projection $\Sigma[Q_R]$, leaf nodes are created in the tree. Then from each pair of joined FDs in the ordered list, a new node is created that has two child nodes pointing to the last node that has a path to the joined leaf or the leaf itself if no such node exists.

5.2.2 Merge join trees

Now that all read actions $Q_R \in \Theta$ have been transformed into join trees, we need to find the subtrees that overlap between them. By finding these overlapping sections, you can see which FDs are joined in multiple read actions. The join trees are merged as follows:

For each join tree $TargetTree_i$ created from the read actions $Q_{\{R,1\}}, \ldots, Q_{\{R,n\}} \in \Theta$, we select all other read actions $Sources_i = (\Theta - Q_{\{R,i\}})$. All read actions $Q_{\{R,j\}} \in Sources_i$ are transformed into ordered lists $SourcesJoined_i$ of pairs of FDs joined ordered by cardinality, the same way as when building the join tree itself. Then, for each join tree $TargetTree_i$, we iterate over all the lists of joined fds in $SourcesJoined_i$ and remove all the joins that have already been made in $TargetTree_i$, and create new join nodes for the joins that have not been made and append them to $TargetTree_i$. This results in n join trees that contain all joined FDs each, but the structure of the merge trees differs, since the order of the join nodes created depends on which tree was merged first. We will denote this list of merged trees as MT.

5.2.3 Nodes selection

From the merged trees MT that have been generated, our objective is to form the collections $Single$ and $Joined$. Each node in a merged tree of MT represents a table in a decomposition, whereas the attributes of the nodes child FDs are the attributes of the table. The goal in selecting nodes is to find nodes with a given ℓ workload inefficiency $\ell_{\Theta}^{U,Total}$ and while having the ℓ highest workload efficiency $\ell_{\Theta}^{J,Total}$. The higher nodes in the merged trees have the highest level of $\ell_{\Theta}^{U,Total}$ and $\ell_{\Theta}^{J,Total}$, since the attributes of the children of a join node are always a subset of the parent. Therefore, it decreases the level of redundancy since fewer FDs will be in the child node.

We apply a greedy algorithm to find the nodes with the highest nodes in the trees for a given maximum level of update inefficiency g. This maximal level of update inefficiency of node v is defined as follows:

$$NU(Q_U, v) = \{X \mid X \rightarrow Y \in \Sigma[Q_U] \wedge X \rightarrow Y \in \sigma[v] \wedge \Sigma \not\models X \rightarrow v\} \tag{5.1}$$

$$\ell_{\Theta}^{update,v} = \max_{Q_U \in \Theta} \max_{X \in NU(Q_U, v)} \ell_X \tag{5.2}$$

The set $NU(Q_U, v)$ represents all redundant LHS of FDs in the update action Q_U if node v was a table. They are selected by finding all the FDs $X \rightarrow Y$ that are in the projection of Q_U and v. From this intersection of FDs, we remove the FD which LHS is a superkey of v, since the key of a table has a redundancy of 1. Finding the maximum update inefficiency $\ell_A^{update,v}$ of v is then the maximum ℓ_X of all FDs in $NU(Q_U, v)$.

Using $\ell_\Theta^{update,v}$, we can select nodes with a specific update inefficiency g and the highest join efficiency from a merged join tree $T \in MT$. We achieve this by starting at all the root nodes of each merged tree. Root nodes will always have a higher or equal join efficiency than their children, since they contain all their joins. If $\ell_\Theta^{update,v} \leq g$ is for a root node, then we return that node. However, if $\ell_\Theta^{update,v} > g$, we recursively check its child nodes until the score is less than g.

This process results in nodes with a maximal update inefficiency of g and the highest join efficiency for each merged tree. To obtain *Single* and *Joined*, we want to select the merged tree that results in the highest join efficiency. This is done by calculating $\ell_\Theta^{J,total}$ for the selected node of each merged tree and selecting the nodes of the merged tree T_{max} with the highest score. The sets *Single* and *Joined* are then defined as follows:

$$Single = \{X \in T_{max} \mid |X| = 1\} \tag{5.3}$$

$$Joined = \{X \in T_{max} \mid |X| > 1\} \tag{5.4}$$

Example 5.1: To demonstrate the process of creating sets *Single* and *Joined* in a comprehensive way, we have the following read actions $\Theta = \{(XYZAB, \emptyset), (XYAB, AB)\}$, with FDs/CCS:

- $X \to YA$, $CARD(X) \leq 1$
- $Y \to Z$, $CARD(Y) \leq 3$
- $A \to B$, $CARD(A) \leq 10$

This will result in the following ordered list: $((X \to YA, Y \to Z), (X \to YA, A \to B))$ for $(XYZAB, \emptyset)$. $X \to YA$ and $Y \to Z$ are first on the list since $CARD(Y) < CARD(A)$. Figure 5.1 shows how the nodes are then created from an ordered list.

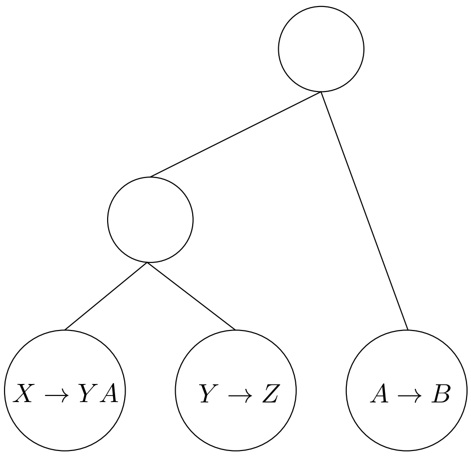

Figure 5.1: Example join tree 1

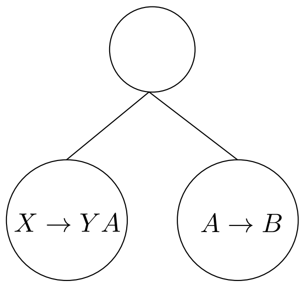

Figure 5.2: Example join tree 2

If we merge the join tree 1 from Figure 5.1 into tree 2 in Figure 5.2, then the merged tree will first join $X \to YA \bowtie A \to B$ and then $X \to YA \bowtie Y \to Z$ since tree B had already made that join. However, if we merge tree 2 into tree 1, there is no node that represents $X \to YA \bowtie A \to B$, since $Y \to Z$ is joined first. Thus, the trees are merged directly on the leaf nodes. Figures 5.3 and 5.4 show the resulting merged trees from the merging in both directions. Although merging tree 1 into tree 2 results in fewer nodes than the other way around, we still want to generate all merge trees since fewer nodes does not mean lower update inefficiency.

The selected nodes in the merged tree differ according to which tree is used. If the FD $A \to B$ is updated and $g = 1$, then from the first merged tree in Figure 5.3, the leaf nodes would be

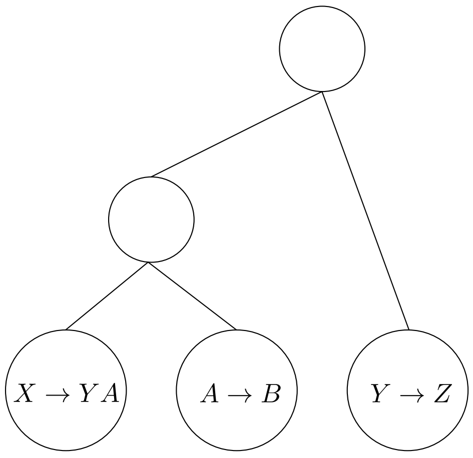

Figure 5.3: Tree 1 merged into tree 2

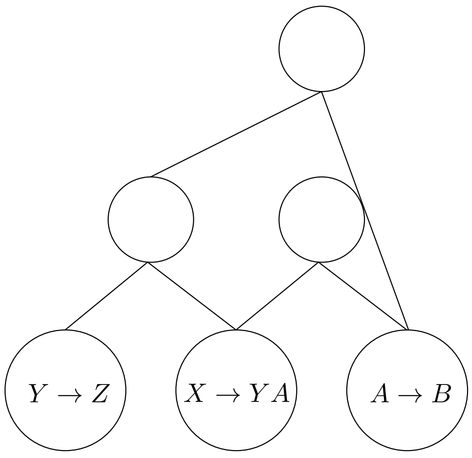

Figure 5.4: Tree 2 merged into tree 1

selected. This is because in all join nodes of this tree the LHS A is never a key and its cardinality is $CARD(A) \leq 10$. In the second merged tree in Figure 5.4 different nodes will be selected. This tree has two root nodes, and therefore some overlapping nodes will be returned. From the top root, the join node $X \to YA \bowtie Y \to Z$ will be selected because it does not contain $A \to B$ and the leaf node of $A \to B$. From the second root, the leaf nodes $X \to YA$ and $A \to B$ are selected. Selecting nodes from these two roots yields the nodes:

- $X \to YA \bowtie Y \to Z$
- $A \to B$
- $X \to YA$
- $A \to B$

This example shows that selecting nodes from multiple roots can yield duplicating FDs between nodes. If two nodes contain the same FD, then the node with the highest join efficiency is selected because all selected nodes already have the requested level of update inefficiency g. If the node that was not selected is a leaf, then its removed. However, if it is a join node, then its children are added to the selected nodes since they can still yield joins that are not made in any other nodes that could yield a higher join efficiency.

5.3 Workload aware algorithms

In this section, we will describe algorithms that create decompositions using the sets *Single* and *Joined*. To understand how the algorithms work, we will first describe the properties of two algorithms from the paper by Link et al. [10]. Our algorithms use their implementations or extensions of their algorithms. We will not describe their normalisation algorithms, but only the properties that are relevant to our work. For a full description of the algorithms, we refer to their work.

The first normalisation algorithm introduced by the authors in their work is called OPT(R, Σ). This is a FD-preserving algorithm that minimises both the join efficiency $\ell^J_{\mathcal{D}}$ and the update inefficiency $\ell^U_{\mathcal{D}}$. It minimises these scores by creating an *atomic cover* Σ_a of FDs Σ [9]. The *atomic cover* are L-reduced FDs with a singleton attribute on the RHS. It introduces new FDs using the resolution rule:

$$\frac{X \to A \; AY \to B}{XY \to B} \tag{5.5}$$

The new FDs created by this rule allows for multiple valid canonical covers. OPT(R, Σ) creates tables using FDs that have the lowest redundancy. To illustrate how this reduces redundancy, we can use the following FDs: $\Sigma = \{V \rightarrow C, E \rightarrow C, CT \rightarrow V, TV \rightarrow E, ET \rightarrow V\}$. An FD-preserving decomposition would result in the tables:

- $R_1 = EC$ with $E \rightarrow C$
- $R_2 = ETV$ with $TV \rightarrow E, ET \rightarrow V$
- $R_3 = CTV$ with $CT \rightarrow V, V \rightarrow C$

Lets say that the $CARD(V) \leq 1000$ and $CARD(E) \leq 10$ this would mean that $\{R_1, R_2, R_3\}$ decomposition would have an efficiency $\ell^U = \ell^J = 1000$, since V is not a key of CTV. If we apply the resolution rule to Σ, we obtain $\Sigma_a = \Sigma \cup CT \rightarrow E$ by applying the rule to $CT \rightarrow V$ and $TV \rightarrow E$. If we then compute the canonical covers, we obtain:

- $\Sigma_1 = \{V \rightarrow C, E \rightarrow C, TV \rightarrow E, ET \rightarrow V, \mathbf{CT} \rightarrow \mathbf{V}\}$
- $\Sigma_2 = \{V \rightarrow C, E \rightarrow C, TV \rightarrow E, ET \rightarrow V, \mathbf{CT} \rightarrow \mathbf{E}\}$

Creating a decomposition for Σ_2 will result in the following tables:

- $R_2 = ETV$ with $TV \rightarrow E, ET \rightarrow V$
- $R_4 = VC$ with $V \rightarrow C$
- $R_5 = CET$ with $CT \rightarrow E, E \rightarrow C$

This decomposition reduces the efficiency to $\ell^U = \ell^J = 10$, since V is no longer redundant but E is. OPT(R, Σ) creates a decomposition from the canoncial cover which results in the lowest redundancy.

The second normalisation algorithm is called HYBRID(R, Σ). It first runs the OPT(R, Σ) algorithm, and then checks each of the decomposed schemata that are in 3NF, but not BCNF. Thus, it selected tables $S \in \mathcal{D}$ where $A \rightarrow B, X \rightarrow Y \in \Sigma[S]$ and $AB \subseteq XY$ with AB not being in the closure of X. If so, the algorithm tries to trade the preservation of the original FD for one with a lower cardinality. This results in the loss of the lower cardinality FD, but does yield a less redundant decomposition.

Using these two algorithms, we have created 3 variations of workload-aware decomposition algorithm. Two of which loosen the 3NF requirement to 2NF, by allowing transitive FDs ($X \rightarrow Y, Y \rightarrow Z$) to be in the same table, and one algorithm that preserves the lost FDs from HYBRID if they are used in a workload.

5.3.1 2NF-FD Preserving algorithm

This first algorithm is an extension of OPT(R, Σ). OPT guarantees a 3NF decomposition. This new algorithm OPT-2NF$(R, \Sigma, \mathit{Joined})$ reduces 3NF to 2NF by allowing transitive dependencies $X \rightarrow Y$ and $X \rightarrow Z$ to exist in the same table, if they are an entry in the set *Joined*. We choose to allow this reduction to 2NF, since the extra redundancy would yield a greater workload join efficiency $\ell_{\Theta}^{J,Total}$.

In Algorithm 1 the implementation is shown. As mentioned first, the decomposition $\mathcal{D}$ is created by the OPT algorithm (line 1). Then a hash map is created with as key the FDs of Σ_a and the values of the tables $S \in \mathcal{D}$ (lines 2-7). For each subset of joined FDs in *Joined* are transformed into tables with the hash map. All attributes of these tables are merged into a new table, and all references in the hash map are updated using the projection of this new table. This overrides the pointers to the original tables (lines 8-13). Finally, all distinct tables are returned from the hash map as the new decomposition.

Algorithm 1 OPT-2NF($R, \Sigma, \mathit{Joined}$)

```
Require: $(R, \Sigma)$ with CC/FD set $\Sigma$ over schema $R$
Ensure: FD-preserving 2NF decomposition $\mathcal{D}$ of $(R, \Sigma)$
 1: $\mathcal{D}, \Sigma_a \leftarrow$ OPT$(R, \Sigma)$
 2: $\mathcal{D}$-map $\leftarrow \emptyset$
 3: for all $S \in \mathcal{D}$ do
 4:     for all $X \to Y \in \Sigma_a[S]$ do
 5:         $\mathcal{D}$-map$[X \to Y] \leftarrow S$
 6:     end for
 7: end for
 8: for all $\Sigma_{table} \in \mathit{Joined}$ do
 9:     $S' \leftarrow \bigcup_{X \to Y \in \Sigma_{table}} \mathcal{D}$-map$[X \to Y]$
10:     for all $X \to Y \in \Sigma_a[S']$ do
11:         $\mathcal{D} - map[X \to Y] \leftarrow S'$
12:     end for
13: end for
14: $\mathcal{D}$-new $\leftarrow$ distinct $S$ in $\mathcal{D}$-map's values
15: return $\mathcal{D}$-new
```

Algorithm 2 HYBRID-FD-Preserving($R, \Sigma, \mathit{Single}, \mathit{Joined}$)

```
Require: $(R, \Sigma)$ with CC/FD set $\Sigma$ over schema $R$
Ensure: Lossless decomposition $\mathcal{D}$ of $(R, \Sigma)$
 1: $\mathcal{D}, \Sigma_a \leftarrow$ OPT$(R, \Sigma)$
 2: for all $S \in \mathcal{D}$ do
 3:     $S = XA$ for some $X \to A \in \Sigma_a$
 4:     if $\exists X' \to A' \in \mathit{Single} \mid XY = X'Y'$ then
 5:         continue
 6:     end if
 7:     for all $Y \to B \in \Sigma_a(YB \subseteq XA \wedge XA \nsubseteq Y^+_{\Sigma[FD]})$ do
 8:         if $\ell_y > \ell_X \wedge \nexists \mathit{Join} \in \mathit{Joined} \mid |\{X \to A, Y \to B\} \cap \mathit{Join}| = 2$ then
 9:             $S_1 = YB, S_2 = Y(XA - B)$
10:             $\mathcal{D} \to (\mathcal{D} - \{(S, \Sigma_a[S])\}) \cup \{(S_1, \Sigma_a[S_1]), (S_2, \Sigma_a[S_2])\}$
11:         end if
12:     end for
13: end for
14: Remove any non-maximal schema from $\mathcal{D}$
15: return $\mathcal{D}$
```

5.3.2 HYBRID-FD Preserving algorithm

The original HYBRID(R, Σ) algorithm allowed FDs to be lost to further decrease the redundancy of a decomposition. This trade-off significantly impacts performance. Therefore, we apply sets *Single* and *Joined* to prevent FDs used in the workload from being lost. In Algorithm 2 we show the implementation. It is nearly the same as the original HYBRID algorithm, except for the extra checks on lines 4 and 8. The if statement on line 4 ensures that no FD in the set *Single* is lost, and the second part of the if statement on line 8 ensures that if both FDs are in *Joined* they are never decomposed.

5.3.3 HYBRID 2NF algorithm

The final algorithm combines the previous two algorithms. It allows the FDs to be lost if they are not in *Single* and *Joined*, and allows tables to merge if the FDs of both tables are in the *Joined* set. The implementation is the same as in Algorithm 1, but instead of running OPT(R, Σ) at the start HYBRID-FD-Preserving$(R, \Sigma, Single, Joined)$ is ran.

Chapter 6

General workload-aware decomposition performance

6.1 Introduction

In this chapter, we test the algorithms that automatically create a decomposition for a given workload. The purpose of this chapter is to show that an algorithm that is tuned to a specific workload has better update $\ell_{\Theta}^{U,total}$ and join efficiency $\ell_{\Theta}^{J,total}$ on joined FDs than a workload unaware algorithm. In addition, we want to show the physical performance in an RDBMS.

6.2 Motivation

With the results of the experiments in this chapter, we hope to show that a decomposition can be automatically created for a given workload. The goal is to show that a decomposition that is tuned to a specific workload has greater update and join efficiency on joined FDs than a workload unaware algorithm. Furthermore, we want to show that not only the theoretical performance improves, but also the physical performance in an RDBMS.

6.3 Experimental design

The baseline for all experiments are the $\text{OPT}(R, \Sigma)$ and $\text{HYBRID}(R, \Sigma)$ decomposition algorithms, since these algorithms already reduce update inefficiency. This allows us to see whether the update efficiency improves for the workload. We choose to reuse the data sets china and ncvoter[1] from the original work. The china data set consists of 262920 rows and 18 columns, and the ncvoter data set has 1024000 rows and 19 columns. Both data sets were originally used for a quantitative study to show the general performance of the authors' algorithms. To create a decomposition from the data sets, we need to obtain the FDs. In the original article, the authors used their own algorithm, called DHyFD, to extract all FDs from the data sets [14]. This algorithm is based on HyFD [12]. In order to be more efficient, we utilized the publicly available implementation[2] of HyFD to extract FDs. However, this did yield a different number of FDs as seen in Table 6.1. This should not hurt the results, but makes the results less comparable to the original quantitative study.

To measure the performance of different workloads in the china and ncvoter data sets, we first need to create workloads. Therefore, we must create a valid join and update queries from the mined FDs. These join queries are made by transforming the set of all FDs Σ into a network graph. Each FD is a node in the graph and an edge is created between two FDs $X \to A, Y \to B \in \Sigma[Q_R]$

[1]`https://www.dropbox.com/sh/28all8zqr637hlr/AAD9_GHf11A8tGiFpdpaqc3ua?dl=0`
[2]`https://github.com/codocedo/hyfd`

	# FDs Original	# FDs Mined
NCVoter	568	477
China	918	840

Table 6.1: Difference # FDs original paper vs. mined

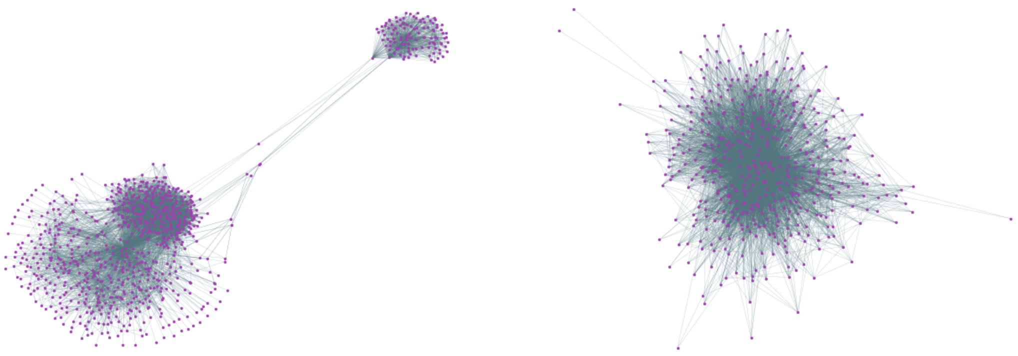
Figure 6.1: Graph of the FDs in the china (left) and ncvoter (right) data sets

when $Y \subseteq XA$ holds. Figure 6.1 shows a visualisation of the graphs created. It shows that many edges are created for all the mined FDs. This happened because the LHS of the mined FDs can have a large number of attributes. Therefore, XA will often be a super set of Y, creating many edges.

Join queries are created by selecting a node's outgoing edges and selecting the neighbour's neighbours until given number of nodes is found. Each FD visited is then an FD in a read action. Update actions are then created as a single FD of the nodes of a read action. Thus, for each action, $(Q_R, Q_U) \in \Theta$ is $Q_U \subseteq Q_A$. Figure 6.2 shows an example of all the selected nodes and the joined edges when creating 128 random read actions with 4 joins each. When comparing Figures 6.1 and 6.2, it can be seen that most FDs are used in queries, but not all edges. This happens because of the nature of mined FDs, since the LHS often contains many attributes which causes many edges to be able to be created.

In the experiments, we are going to control two main parameters to measure the effectiveness of the created decompositions. These are: "the number of actions to generate" and "the fractions of

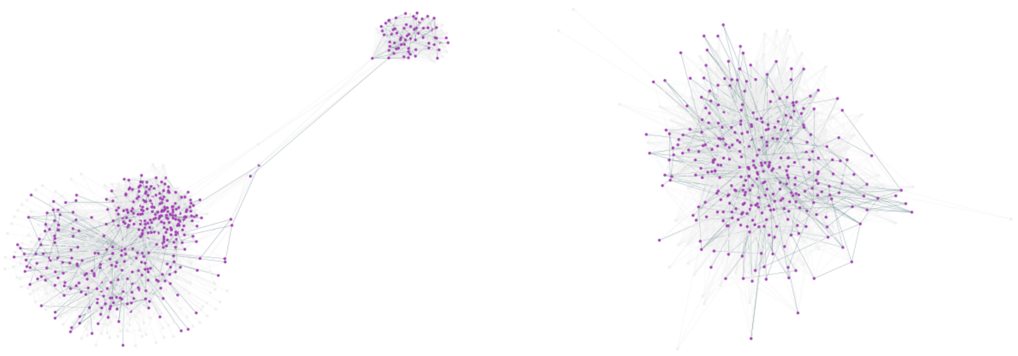
Figure 6.2: Visualization of the nodes visited with 128 queries with 4 joins each

actions that are updates". The experiments consist of two parts. First, statically, by analysing the total workload update inefficiency $\ell_{\Theta}^{U,total}$ and the join efficiency $\ell_{\Theta}^{J,total}$ of the created workloads. Second, by executing the joins of a given workload in SQL.

6.4 Static analysis

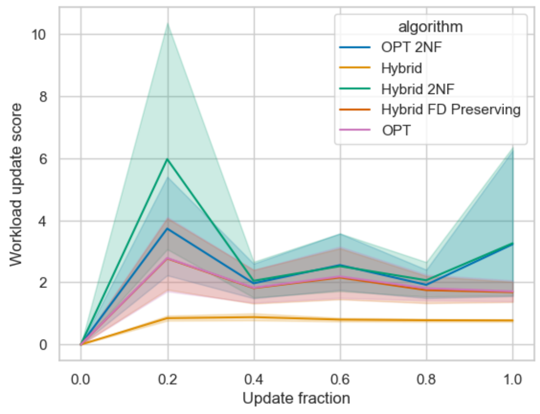

Figure 6.3: Line plot of mean $\ell_{\Theta}^{U,total}$ for each update fractions for china data set

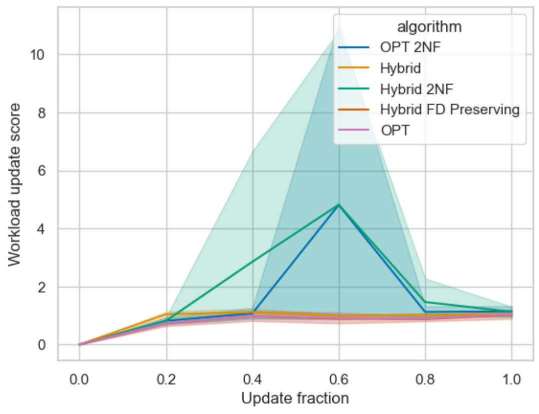

Figure 6.4: Line plot of mean $\ell_{\Theta}^{U,total}$ for each update fractions for ncvoter data set

In this section, we describe the results of the static analysis. For each combination of parameters, 10 different workloads are created and the mean result of the created workload is shown. Figures 6.3 and 6.4, show this mean workload update score $\ell_{\Theta}^{U,total}$ for different fractions of updates for total queries. The gradient area behind the lines shows the 95th percentile confidence interval of the bootstrap distribution for each algorithm. It can be observed that, in general, the $\ell_{\Theta}^{U,total}$ of OPT, HYBRID, and HYBRID FD preserving are close to constant for all update fractions with an optimal score around 1. However, the 2NF algorithms for OPT and HYBRID do have some seemingly random increases to slightly higher levels of redundancy. The confidence level on these peaks is also low due to having both high and low scores for these parameters. Overall, the score in general is still mostly around 1 for these algorithms.

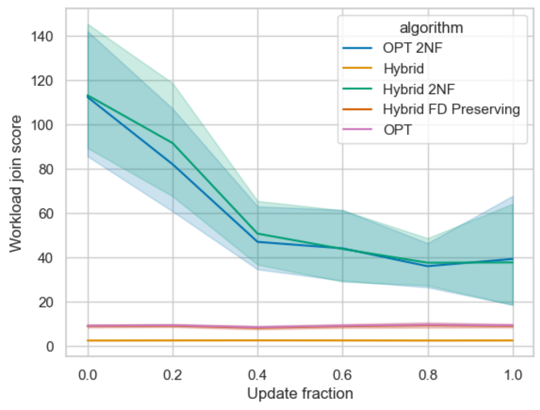

Figure 6.5: Line plot of mean $\ell_{\Theta}^{J,total}$ for each update fractions for china data set

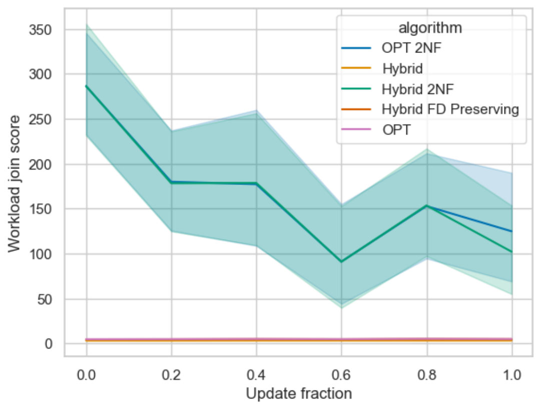

Figure 6.6: Line plot of mean $\ell_{\Theta}^{J,total}$ for each update fractions for ncvoter data set

The total efficiency of joining $\ell_{A}^{J,total}$ for generated workloads is strongly affected by the percentage of updates, as seen in the figures 6.5 and 6.6. For the 2NF algorithms, there is a clear downward trend, visible when increasing the percentage of updates. For example, with OPT 2NF $\ell_\Theta^{J,total}$ decreases from 112 for 0% updates to 39 for 100% update actions with the china data set. The other algorithms still have a low join score, but this is expected, since they optimise the decomposition for update efficiency. The OPT and Hybrid FD Preserving perform the same, with a $\ell_\Theta^{J,total}$ of 9 for the china data set.

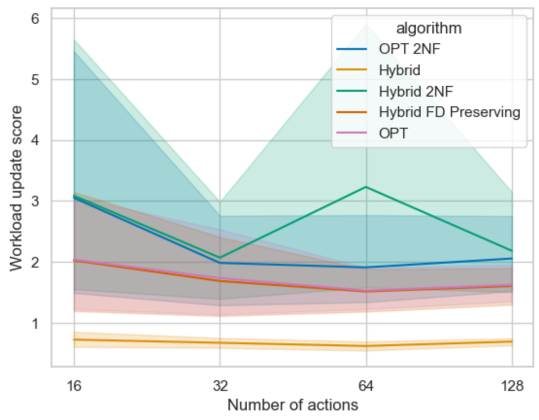

Figure 6.7: Line plot of mean $\ell_\Theta^{U,total}$ for each # actions for china data set

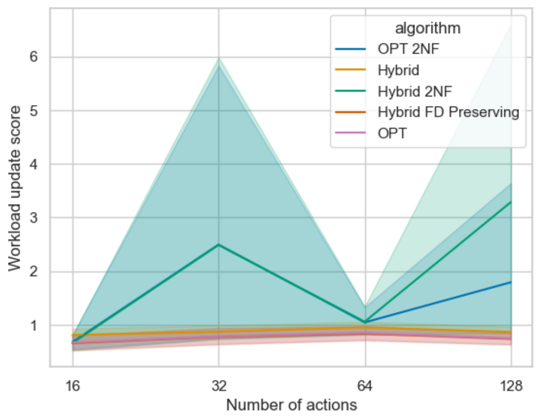

Figure 6.8: Line plot of mean $\ell_\Theta^{U,total}$ for each # actions for ncvoter data set

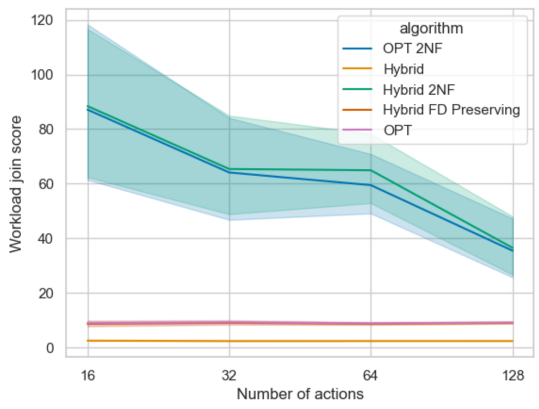

Figure 6.9: Line plot of mean $\ell_\Theta^{J,total}$ for each # actions for china data set

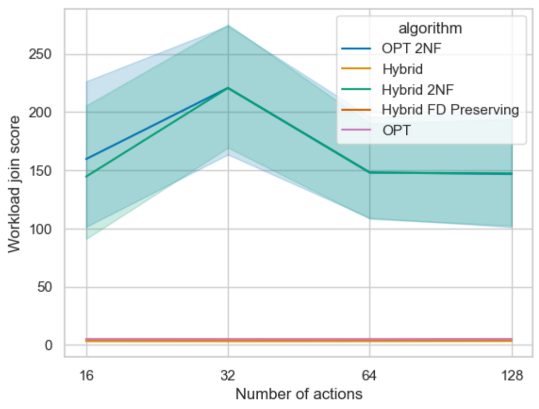

Figure 6.10: Line plot of mean $\ell_\Theta^{J,total}$ for each # actions for ncvoter data set

To see the impact of increasing the number of actions on the created decompositions, we measure the same update inefficiency $\ell_\Theta^{U,total}$ and the join efficiency $\ell_\Theta^{J,total}$, but this time for the number of actions. Figures 6.7 and 6.8, show the update inefficiency $\ell_\Theta^{U,total}$. Once again do the 2NF algorithms have the highest level of redundancy on the updated FDs, but it is only slightly worse. The number of actions does not appear to affect the update inefficiency $\ell_\Theta^{U,total}$. Especially, when we look at the confidence level, there is a high variance in $\ell_\Theta^{U,total}$ for the number of actions.

The join efficiency $\ell_\Theta^{J,total}$ seems to be affected by the number of actions. In Figure 6.9, we can clearly see a downward trend in efficiency for the decompositions created by the 2NF algorithms. For OPT 2NF, the efficiency decreases from 87 to 35 for the china data set. Surprisingly, this pattern does not appear for the ncvoter data set in Figure 6.10. The 2NF algorithms have a constant $\ell_\Theta^{J,total}$ of ∼150 with a peak for 32 actions of 220.

The fraction of updates and the number of actions both heavily affect the decomposition created by the 2NF algorithms. To understand which parameter has the greatest impact, we analyse both

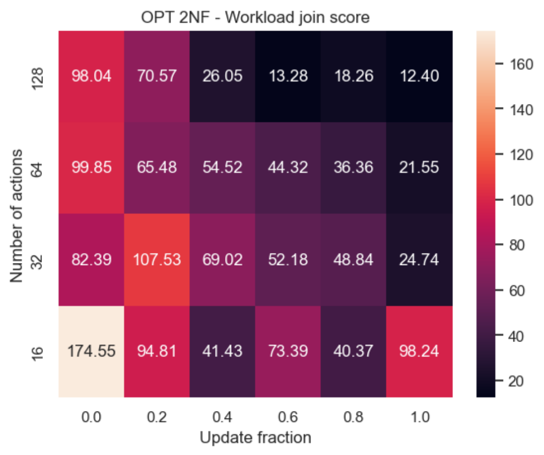

Figure 6.11: Heat map of mean $\ell_{\Theta}^{J,total}$ for each # actions and update fraction for china data set

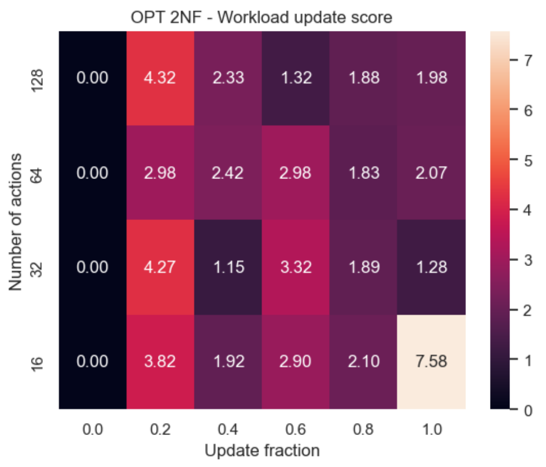

Figure 6.12: Heat map of mean $\ell_{\Theta}^{U,total}$ for each # actions and update fraction for china data set

parameters separately for OPT 2NF. In Figure 6.11 shows the join efficiency $\ell_{\Theta}^{J,total}$ as a heat map. In this figure, we can see that the join score decreases by increasing the update fraction for almost all values of the number of actions. However, increasing the number of actions does not always yield a lower join efficiency.

A heat map for the update inefficiency $\ell_{\Theta}^{U,total}$ of OPT 2NF can be seen in Figure 6.12. Here we can see that both parameters do not have much of an impact. The only clear observation for a low update fraction of 0.2 is that the update inefficiency is slightly higher for all number of actions.

6.5 Performance simulation

In the second experiment, we measured the impact of workload-aware decompositions in an RDBMS system. Specifically, we run the experiments on a computer with an Intel i7-9750H and 8 GB of RAM. For the RDMS we used MySQL version 3.8 and for the data we used the china data set. In this experiment, we measure the difference in the joining performance of the different decompositions. Tables in MySQL are created from the attributes of the FDs in a table in a decomposition. A clustered index is created of the attributes in the LHS the FD that has all the attributes in its closure.

The queries are generated in the same way as in the previous section, but only with 16 actions and update fractions of 0.2 and 0.6. Each query contains 5 FDs. For each table S containing FDs $X \to Y \in \Sigma[S]$, we define the following sets to create the queries:

$$key = \{X \mid X \to Y \in \Sigma[S] \wedge \Sigma[S] \models X \to S\} \tag{6.1}$$

$$a = (\bigcup_{X \to Y \in \Sigma[S]} XY) - key \tag{6.2}$$

Using these set we create the following SQL queries:

```
SELECT *
FROM
    { Table_1 } if attributes in key are the primary key of Table_1 else
    { SELECT
        key_1, ..., key_n,
        MIN(a_1), ..., MIN(a_n)
```

```
      FROM Table_1 GROUP BY key_1, ..., key_n }
NATURAL JOIN ...
NATURAL JOIN
    { Table_n } if attributes in key are the primary key of Table_n else
    { SELECT
        key_1, ..., key_n,
        MIN(a_1), ..., MIN(a_n)
      FROM Table_n GROUP BY key_1, ..., key_n }
```

We group the attributes before joining since joining on attributes that appear more than once would result in a join explosion. For example, if we have the following decomposition:

- $Table_1(AB)$ with FD $A \to B$ and CC $CARD(A) \leq 1$
- $Table_2(XYB)$ with FDs: $XY \to B, B \to Y$ and CCs: $CARD(X) \leq 1, CARD(B) \leq 10$

If we would like to select ABY then we would join $AB \bowtie BY$. Each selected row would be duplicated 10 times since $CARD(B) \leq 10$. Therefore, we group the attributes in B and take the first value of Y to avoid duplication. We can take the first value, since the FD $B \to Y$ tells us that Y is the same for all B.

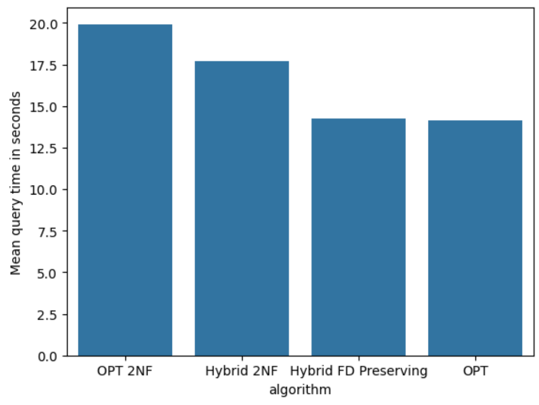

Figure 6.13: Mean join query time on china data set

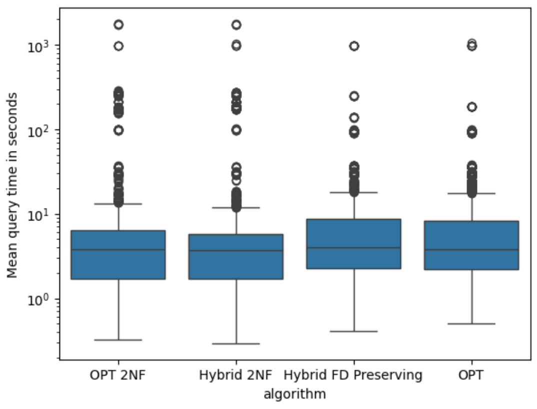

Figure 6.14: Box plot showing the median and interquartile range of the join query time

In Figure 6.13 the mean join times of the queries are shown. 2NF algorithms perform significantly worse than OPT. The dependency-preserving hybrid algorithm performs the same as OPT. Figure 6.14 also shows the join query time, but as a box plot of the interquartile range (IQR) in the blue area. The dots show the query times outside $Q3 + 1.5 \cdot IQR$. This figure shows that most queries of the 2NF algorithms take less time, but that there are some queries that take significantly longer.

In static analysis, the join efficiency $\ell_{\Theta}^{J,total}$ was mainly affected by the fraction of update queries. To see whether the update fraction affects the join efficiency in the same way, we divide the box plot of Figure 6.14 by the update fraction in Figure 6.15. Surprisingly, the median join time is slightly slower for the 0.2 update fraction. However, this is not due to more joins, as the lower join efficiency does join less tables as can be seen in Figure 6.16.

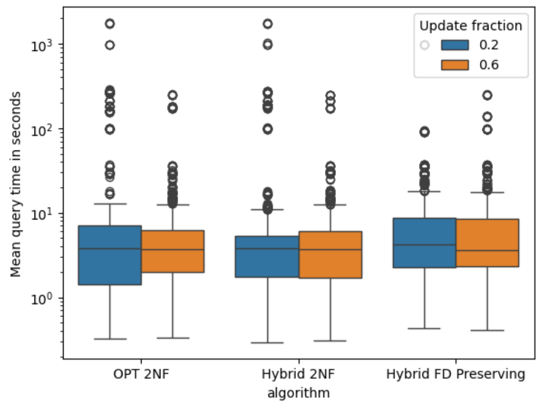

Figure 6.15: Box plot showing the median and interquartile range of the join query time split by update fraction

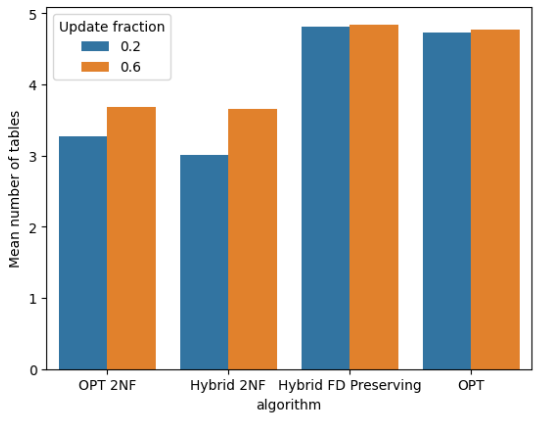

Figure 6.16: The mean number of tables tables per fraction

6.6 Discussion and findings

Static analysis shows that 2NF algorithms generally have a higher $\ell_{\Theta}^{J,total}$. Increasing the fraction of actions that are update actions decreases the join efficiency. This is expected behaviour since they both use the *Joined* set to create the tables in 2NF. This set only brings the FDs together when it results in an update inefficiency of 1. Therefore, if there are more actions that contain updates, then there are fewer 2NF tables. This also explains why the efficiency join decreases only for a greater number of actions if the update fraction is not 0.0.

2NF algorithms generally have greater update inefficiency $\ell_{\Theta}^{U,total}$. The fact that the number of actions and the update fraction have little impact on $\ell_{\Theta}^{U,total}$ can also contribute to the way the *Joined* set is created. All subsets of FDs in *Joined* have at most an update inefficiency of 1. Therefore, the update inefficiency should not increase for any increase in the number of actions or the update fraction. The reason why the 2NF algorithms have a greater $\ell_{\Theta}^{U,total}$ than 1 is due to the merging of tables. The set *Joined* exclusively comprises the FDs that require joining, while the tables generated by OPT or HYBRID may contain multiple FDs. If two tables are merged using *Joined*, additional redundancy may arise as a result of other FDs present in the table generated by OPT or HYBRID. This specific phenomenon occurs especially often for mined FDs, since they have a large number of overlaps attributed between FDs.

The Hybrid FD Preserving algorithm has the same $\ell_{\Theta}^{J,total}$ and $\ell_{\Theta}^{U,total}$ as OPT for all parameters. This is the expected result, as none of the FDs that are used in any of the actions of A can be decomposed. This does mean that there are slightly greater $\ell_{\Theta}^{J,total}$ and $\ell_{\Theta}^{U,total}$ than Hybrid, but none of the FDs used are lost.

The physical performance of the decompositions shows that most queries perform better in the 2NF decompositions, as seen in Figure 6.14. However, it is also evident that these decompositions exhibit a higher number of extreme query times, resulting in lower average performance compared to OPT. This can be contributed to the grouping that needs to be done before joining the tables when a joined FD has a greater cardinality than 1. The creation of *Joined* attempts to minimise this redundancy, but once again merging the tables with multiple FDs can introduce extra redundancy that was not in the subsets of *Joined*. Especially in these mined FDs it is hard to judge the algorithms performance, due to the extreme number of FDs with overlapping attributes.

The difference in join performance between queries on the 2NF algorithm's decompositions shows that the performance does not change much. This is surprising, since fewer tables are joining, as seen in Figure 6.16. Random selection of FDs in a workload can explain this, as the cardinality of the joined FDs can differ greatly between worklaods. The FDs in the workload should be the same to measure this more accurately.

To conclude, the 2NF algorithms greatly increase $\ell_{\Theta}^{J,total}$ and only slightly hurt $\ell_{\Theta}^{U,total}$. In general, the performance of the join queries in SQL also seems to improve. However, if there are many overlapping attributes between FDs, such as in the FDs mined, then the created decompositions can significantly hurt join performance due to the need for deduplication. The random selection of FDs in workload also causes outliers in results due to differing levels of cardinality in the FDs.

Chapter 7

Micro-benchmark of generated decompositions

7.1 Introduction

Chapter 6 showed the syntactic performance of the decomposition algorithms and with that gave insight into the workload join and update efficiency on a high-level. However, due to the syntactic nature of the mined FDs and the randomly generated workloads, it was hard to understand how the algorithms would perform in a real-word scenario. Therefore, we are going to analyse the created decompositions on manually selected functional dependencies and hypothetical workload, with the goal of understanding how closely the generated decompositions fit the workload and how accurately the workload metrics represent the actual update and join efficiency.

7.2 Workload

We chose to use the ncvoter data set for this case study. Table 7.1 has the manually selected FDs and the cardinalities of the LHS attributes.

The hypothetical workload consists of the following four actions:

- **Action U1**: A voter has changed their last name:
  The last_name is changed for download_month, voter_id

- **Action U2**: A voter has moved to a new address:
  The zip_code, street_address is changed for a download_month, voter_id

- **Action U3**: A phone number is changed for a voter:
  The full_phone_num is changed for a download_month, voter_id

Table 7.1: Manually selected FDs from ncvoter data set

Functional Dependency	**Cardinality**
download_month, register_date, voter_id → age, full_phone_num, street_address, zip_code	1
download_month, voter_id → name_prefix, first_name, middle_name, last_name, name_suffix, gender, race, voter_reg_num, birth_place, ethnic	2
street_address, zip_code → city	208
city, zip_code → state	8637

- **Action J1**: We want to send a letter to voters addressing them by their full name and gender: Selection of the fields: street_address, zip_code, city, state, name_prefix, first_name, middle_name, last_name, name_suffix, gender

Since none of the the selected FDs are contained in another FD we are only going to create decompositions with OPT and OPT 2NF. The HYBRID algorithms would not differ from the OPT algorithms for these FDs.

7.3 Decomposition

The OPT algorithm works by first creating the atomic cover of the FDs. In this scenario, this means that all L-reduced FDs with a singleton attribute are created on the right-hand side. Thus, all FDs are simply split in terms of their RHS attributes. Since there are no redundant FDs in the atomic cover, a table is created for each RHS attribute. Figure 7.1 shows the tables visualised using our tool. Here we can see that there are edges between all tables with download_month, register_date, voter_id and download_month, voter_id as key. In addition, there is no edge to the tables with street_address, zip_code as keys, since the RHS attributes of download_month, register_date, voter_id → street_address, zip_code are split between tables.

Our OPT 2NF algorithm merges the tables when used in the same query. To reiterate, we joined the following attributes: street_address, zip_code, city, state, name_prefix, first_name, middle_name, last_name, name_suffix, gender. Figure 7.2, shows the tables created by OPT 2NF with a maximum update inefficiency of 1. The notable tables are:

- r_1: city, state, zip_code, street_address
- r_2: download_month, voter_id, register_date, gender, name_prefix, middle_name, zip_code, street_address
- r_3: download_month, voter_id, last_name

The FDs street_address, zip_code → city and city, zip_code → state are merged into the bottom node of Figure 7.2. Action U2 tells us that street_address, zip_code → city can be updated and therefore should be a key of its table, and action J1 is the reason why the FDs are merged. Tables r_2 and r_3 have not been merged since action U1 updates the last_name, thus merging the tables would make last_name redundant. However, there is another observation that the attributes first_name, and name_suffix are not contained in r_2 or r_3 even though they are joined in J1. To illustrate how this can happen, we visualise a small section of the join tree in Figure 7.4. Here, you can see that last_name is the bottom node. This causes the join efficiency of the join nodes above it to have an efficiency of 2, which means that all left nodes will have their own table. Therefore, name_suffix and first_name are not included in r_2 and r_3.

If we increase the maximum join inefficiency to 2 then we obtain the tables as seen in Figure 7.3. Here, r_1 and r_2 have been merged, and name_suffix and first_name are also in the same table. The update inefficiency of that merged table increases because last_name now has redundant values.

Table 7.2: Efficiency results of the different different decompositions

	OPT	OPT 2NF u=1	OPT 2NF u=2
$\lvert\mathcal{D}\rvert$	16	11	8
$\ell^J_{\mathcal{D}}$	1	8637	8637
$\ell^U_{\mathcal{D}}$	1	8637	8637
$\ell^{J,total}_{\mathcal{D}}$	16	8650	8647
$\ell^{U,total}_{\mathcal{D}}$	16	8650	8647
$\ell^{J,total}_{\Theta}$	10	8644	8641
$\ell^{U,total}_{\Theta}$	1.66	1.33	1.66

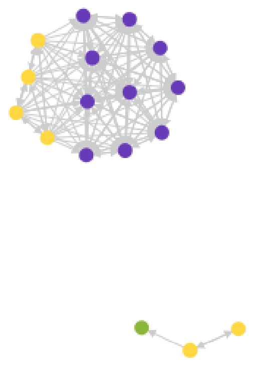
Figure 7.1: ncvoter decomposed using OPT

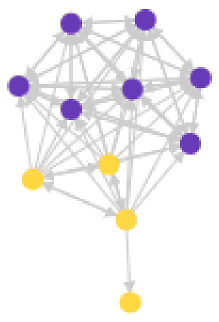
Figure 7.2: ncvoter decomposed using OPT 2NF max update = 1

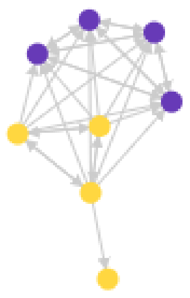
Figure 7.3: ncvoter decomposed using OPT 2NF max update = 2

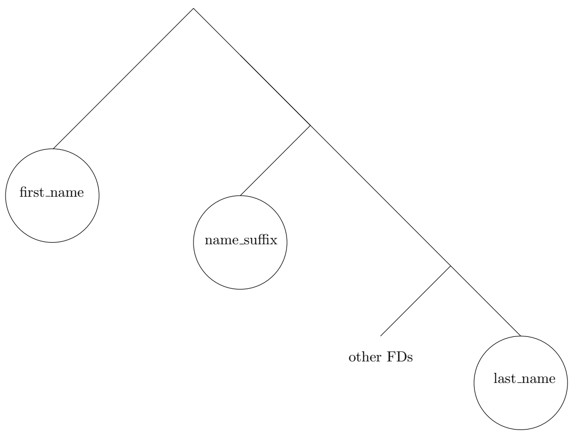

Figure 7.4: Section of the join tree for a query on the ncvoter FDs

With these three decompositions, the metrics workload join efficiency $\ell_{\Theta}^{J,total}$ and workload update inefficiency $\ell_{\Theta}^{U,total}$ can be evaluated to determine how effective they are in measuring the performance of a decomposition. In Table 7.2 the original metrics of ℓ-join efficiency $\ell_{\mathcal{D}}^{J}$, ℓ-update inefficiency $\ell_{\mathcal{D}}^{U}$, total level of join efficiency $\ell_{\mathcal{D}}^{J,total}$, and total update inefficiency $\ell_{\mathcal{D}}^{U,total}$ are shown with our metrics for the decompositions.

The level of ℓ-update inefficiency and ℓ-join efficiency has increased greatly in OPT 2NF decompositions, due to the redundant FD city, zip_code $\rightarrow$ state in table r_1. The total join and update scores $\ell_{\mathcal{D}}^{J,total}/\ell_{\mathcal{D}}^{U,total}$ of OPT 2NF decompositions are also higher for the same reason. However, when increasing the maximum update inefficiency from 1 to 2, it also decreases the total join efficiency $\ell_{\mathcal{D}}^{J,total}$ and the total update inefficiency $\ell_{\mathcal{D}}^{U,total}$ from 8650 to 8647. This is due to join-supportive attribute subset $JS_{\mathcal{D}}^{(R,\Sigma)}$ of equation 4.1 that is used to calculate the total scores. It contains all the unique LHS attributes in each table S in a decomposition $\mathcal{D}$. Therefore, a decomposition with a lower redundancy but more tables can have a greater total scores.

The workload join efficiency $\ell_{\Theta}^{J,total}$ suffers from the same problem as the total efficiency of the joins. Having more tables will also increase this score even with a lower level of redundancy. This also means that more tables can be joined when they have a lower value $\ell_{\Theta}^{J,total}$. For example, to query J1 we need to join 5 tables in the decomposition created for a maximum update redundancy of 1 and 2 tables for a maximum update redundancy of 2.

The results also show that the workload update inefficiency $\ell_{\Theta}^{U,total}$ increases with the number of tables with the same redundancy. For the decomposition created by OPT we obtain $\ell_{\Theta}^{U,total} = \frac{5}{3} = 1.66$, since we need to update 5 tables with no redundancy to update 3 FDs. The OPT 2NF algorithm correctly enforced the maximal redundancy in both decompositions. In the first decomposition, the redundancy is $\ell_{\Theta}^{U,total} = \frac{4}{3} = 1.33$, since all tables have a redundancy of 1 and zip_code, street_address are in the same table, we only have to update 4 tables to update the 3 FDs. The other decomposition created by OPT 2NF also has $\ell_{\Theta}^{U,total} = \frac{5}{3} = 1.33$, but this time due to the increase in redundancy of one of the updated tables to 2.

7.4 Discussion and findings

From this small case study, we found that OPT 2NF (and indirectly other new algorithms) creates tables that do not exceed the given maximum level of redundancy for updated FDs and greatly increase the join efficiency for the joined FDs. However, the ordering of the join tree can cause not-updated FDs that are joined together not to merge into a single table, which could lead to more joins to be made than needed. One way to prevent this is by pushing the updated FD further up the tree.

The second major observation made in this case study is that the number of tables strongly impacts both the total join/update efficiency and the workload join/update efficiency. Normally, this is not an issue since both of them represent the sum of the redundancy of all tables, in general, and the ones used. However, when the number of tables increases due to splitting of FDs on their RHS attributes, then the join efficiency increases without creating more redundancy in the tables. Where there is an increase in $\ell_{\Theta}^{J,Total}$, one would assume that there are fewer joins in the query. However, when the number of tables is increased, naturally, the number of joins will also increase even though the workload join score $\ell_{\Theta}^{J,Total}$ also increases. For the workload update inefficiency $\ell_{\Theta}^{U,Total}$, this increase in tables does not make the results harder to interpret. If the inefficiency grows with the number of tables, it still represents that more rows need to be updated; only now are the rows split between tables instead of redundant values in a single table.

Chapter 8

Future work

The main thing missing in evaluating a decomposition for a given workload are metrics that represent the amount of duplicate elimination needed. Right now, the join efficiency informs us about the redundancy introduced, and thereby the fewer joins that have to be made, but joins or simple reads on the redundant FDs can significantly impact performance. Having such a metric would also significantly enhance the cost function for selecting the join nodes of the decomposition decomposition algorithms.

The workloads in Chapters 6 and 7 are not realistic. A case study of a real-world database with its workload of join and updates would provide much inside information on the performance of the generated decompositions and the insights of the join and update efficiency metrics. Furthermore, simulation of real-world schemata and workloads also helps in the assessment of decomposition algorithms. In Appendix A, we have already explored some properties of schemata and workload simulation. The creation of workload simulation tools can help in the design process of decomposition algorithms.

Chapter 9

Conclusions

The introduction of a workload into the total join efficiency $\ell_{\mathcal{D}}^{J,total}$ and total update inefficiency $\ell_{\mathcal{D}}^{U,total}$ helps provide information on actual usage. Whereas previously every removal or introduction of redundancy would alter the metrics evenly, right now practitioners are able to understand how the change in redundancy will impact a join or update action. Furthermore, have we shown that we can measure the redundancy of a single read or update action with the metrics $\ell_{Q_R}^{J,Total}$ and $\ell_{Q_U}^{U,Total}$, these metrics are especially interesting when optimising specific actions or understanding why a change in decomposition altered its performance.

The micro-benchmark in Chapter 7 showed that decreasing $\ell_{\theta}^{J,Total}$ does not always imply decreasing the number of joins. Decomposing tables with little redundancy or creating new tables by splitting FDs on their RHS attributes can also increase $\ell_{\theta}^{J,Total}$, due to all tables contributing a minimum redundancy of 1 to the metric. Furthermore, it does not capture the need for duplicate elimination when joining redundant values as seen by the outliers in Figure 6.14.

The algorithms introduced in this work show that we can automatically create decompositions with increasing redundancy of the joined values without increasing the update inefficiency. The performance of the decompositions created by the algorithms in SQL showed that most queries improve in performance. However, the large amount of redundancy created by allowing transitive FDs in the same table in the 2NF algorithms did show that duplicate elimination should be taken into account, when automatically generating decompositions.

The most important lesson learnt in this work is that functional dependencies together with both Cardinality Constraints and a workload of reads and updates are important tools when decomposing a relational database. Practitioners can use these properties to understand how a level of normalisation will effect their database, both on the impact on join and update performance, but also the impact on storage.

Bibliography

[1] Stack overflow developer survey 2021. 1

[2] William Ward Armstrong. Dependency structures of data base relationships. In *IFIP congress*, volume 74, pages 580–583. Geneva, Switzerland, 1974. 7

[3] Linked Data Benchmark Council. LDBC Social Network Benchmark benchmark version 2.2.3. `https://ldbcouncil.org/ldbc_snb_docs/ldbc-snb-specification.pdf`. 40, 43

[4] Transaction Processing Performance Council. TPC-C benchmark revision 5.11.0. `https://www.tpc.org/tpcc/`. ix, 39, 40

[5] Transaction Processing Performance Council. TPC-E benchmark revision 1.14.0. `https://www.tpc.org/tpce/`. 39, 41

[6] Transaction Processing Performance Council. TPC-H benchmark revision 3.0.1. `https://www.tpc.org/tpch/`. 39

[7] Chris J Date. *Database design and relational theory: normal forms and all that jazz.* Apress, 2019. 1, 7, 8

[8] Statista Research Department. Total data volume worldwide 2010-2025, Sep 2022. 1

[9] Henning Koehler. Finding faithful boyce-codd normal form decompositions. In *Algorithmic Aspects in Information and Management: Second International Conference, AAIM 2006, Hong Kong, China, June 20-22, 2006. Proceedings 2*, pages 102–113. Springer, 2006. 19

[10] Sebastian Link and Ziheng Wei. Logical schema design that quantifies update inefficiency and join efficiency. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1169–1181, 2021. 1, 11, 12, 13, 15, 16, 19

[11] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983. 6, 8

[12] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833, 2016. 23

[13] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. *Don't Focus on Big Data; Focus on the Data That's Big*, 2, 2017. 1

[14] Ziheng Wei and Sebastian Link. Discovery and ranking of functional dependencies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1526–1537. IEEE, 2019. 23

Appendix A

Workload Simulation

A.1 Introduction

In this chapter, we are going to simulate stable workload patterns such that we can validate normalisation algorithms in multiple scenarios. We choose to simulate the workload patterns, since we found no public data was found, and to test algorithms in a more generalisable manner.

We will initially explore some background information on how other database simulations operate. Specifically, we will look at how they create data for a given schema. What rules apply for generation and how realistic the simulations are. Based on this background information, we will describe our solution in two sections. First, how to generate realistic schemata with functional dependencies and table properties. Second, how to generate realistic workloads on a generated schema.

This chapter also describes how schema and workload simulation are implemented in a tool. Using this tool, we do another experiment to test the decomposition algorithms in multiple scenarios. This should give more insight into real world performance.

A.2 Motivation

In the experiments in Chapter 6, we quantitatively analysed the performance of multiple normalisation algorithms based on the generated workloads from the mined FDs. However, the generated workloads and the extracted FDs and data set are not representative of real-world data. Therefore, we want to create tooling that can generate realistic FDs and workloads based on chosen distributions.

A.3 Background

We explored multiple simulations from two councils. First, we have benchmarks from the Transaction Processing Performance Council (TPC). The Transaction Processing Performance Council (TPC) is a non-profit organisation that defines industry-standard benchmarks and metrics for evaluating the performance and scalability of database systems. The benchmarks we reviewed are TPC-C [4], TPC-H [6], and TPC-E [5]. Each benchmark simulates different systems. TPC-C and TPC-E are both OLTP benchmarks where TPC-E model a more complex system. TPC-H is an OLAP benchmark with mostly ad hoc queries.

The benchmarks group their queries into transactions. Each transaction then has a frequency, weight, and whether it is read, written, or both transactions. The frequency and weight are defined abstractly as light, medium, or heavy. Another property which is less explicitly mentioned in execution intervals. In the TPC-H benchmark, all the updates are batches of update queries. This simulates the copying of data from OLTP databases to a datawarehouse.

The three TPC benchmarks have a scaling factor SF that is used to generate the initial data for the tables and the number of queries to execute during the benchmarks. However, the TPC-C and TPC-E benchmarks also define their initial cardinality as the sum of specific core tables. In the TPC-E benchmarks their categorise their tables to explain these generation rules. The categories are fixed tables, scaling tables, and growing tables. They defined these categories as follows:

- Fixed tables: These tables always have the same number of rows regardless of the database size (e.g. countries, enums)
- Scaling tables: These tables each have a defined cardinality that has a constant relationship to the scaling factor (e.g. customers, districts)
- Growing tables: These tables each have an initial cardinality that has a defined relationship to the cardinality of the scaling factor. However, the cardinality increases with new growth during the simulation. (e.g. orders, posts, comments)

For example, in figure A.1 can see the relationship and their cardinality for the TPC-C benchmark. Warehouse, District, Customer, and Stock are the scaling tables. Order, Order-Line, New-Order, and History are growing tables, and Item is a fixed table. By assigning tables to these categories, we can generate workloads more realistically, which we will discuss in the next chapter. Moreover, it also supports the generation of foreign keys by defining rules for these table categories. These rules are that fixed tables can only have foreign keys to other fixed tables. Scaling tables can only have foreign keys to other scaling tables or fixed tables. Growing tables can have foreign keys to tables of all categories, but there must exist a path to a scaling table.

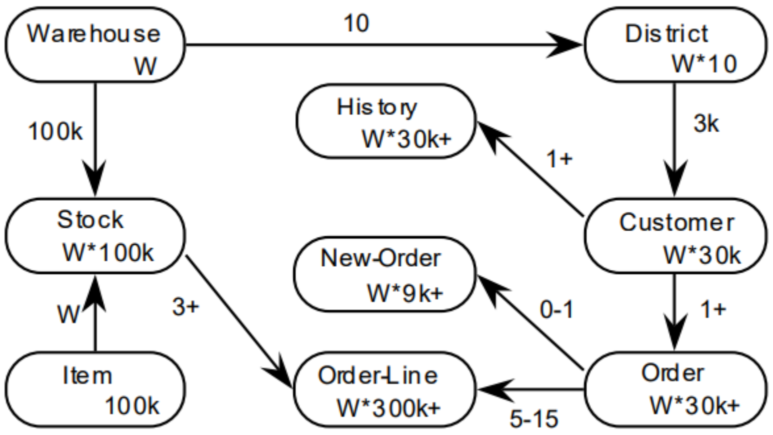

Figure A.1: TPC-C model [4]

The second council from which we reviewed the benchmarks was the Linked Data Benchmark Council (LDBC). This is also a non-profit organisation which defines benchmarks, but they focus on defining graph benchmarks. For this council, we review its Social Network Benchmark (SNB) [3]. As the name suggests, this benchmark simulates the behaviour of a social network. However, where this benchmark differs from the TPC benchmarks is that it contains two different types of workloads. Namely, the interactive and business intelligence workload. In the interactive workload, the number of complex read-only queries is chosen based on a given scaling factor. Since some queries scale worse as the database size grows, the number of executions may decrease, and other queries may be executed more to keep the time for each query equal. For each complex read instance, a sequence of short reads is designed. There are two kinds of short read sequences: Person-oriented and Message-oriented. Depending on the type of complex read, one of them is selected. Each sequence consists of a set of short reads that are issued in succession. For each short read sequence that is run, there is a decreasing probability that another sequence will be triggered. The insert operations create a new node and connect it to existing nodes or create a new edge between two existing nodes. Similarity, do the delete operations remove nodes with all their edges or remove an edge between two nodes.

The business intelligence workload consists of 20 read-only queries. Most of these queries are join-heavy operations on many-to-many relationships. Additionally, they include filtering, grouping, aggregation, and sorting operators.

A.4 Schema Simulation

A.4.1 Modelling

Normalisation algorithms need functional dependencies as input. For modelling tables in a schema, it is mainly important that they represent FDs. Therefore, we assume that all tables in a schema are in 3NF, so that no transitive dependency is allowed in the table. Even though it is possible to express the workloads in terms of the functional dependencies, e.g. $A \rightarrow B$ is selected x times. We choose to model the schema as a network graph. Tables in a database will represent the nodes in the graph and the edges the foreign keys between the tables. When expressing a read query, we can then refer to the edges that are used in the join and easily measure the cardinality of the joined attribute.

The nodes in the graph will have the following properties:

- Name: The name of the table in a schema
- Cardinality: The number of rows in of table
- Columns: The names of the columns
- Keys: Tuples of columns which represent its keys
- FDs: all functional dependencies in the node

The edges have a source and a target, where the source is the table with the foreign key pointing to the target table.

A.4.2 Generation

We want to test our future algorithm on as realistic data as possible, since testing on real datasets is not possible due to the lack of publicly available workload data. In Section A.3 we discuss benchmarks from TPC and LDBC. These councils tried to mimic realistic workloads. Therefore, we tried to apply properties from these benchmarks to generate schemata. Since each of these benchmarks simulates a specific scenario, we had to generalise some of the rules that they apply. The main property for schema generation that we use are the table categories from the TPC-E [5] benchmark. To reiterate, these categories are fixed, scaling, and growing tables. We use these categories to generate edges and the initial cardinality of each node.

The rules for generating edges are:

- **Fixed nodes** have no outgoing edges or only edges to other fixed nodes.
- **Scaling nodes** have outgoing edges to fixed or scaling nodes.
- **Growing nodes** can have edges to nodes of all categories, but there must exist a path to at least one scaling node.

The initial cardinality of each of the nodes is then based on its category, neighbours, and given scaling factor. The cardinality of each type of node is then defined as follows:

- **Fixed nodes**: Cardinality is a constant value taken from a given distribution.

- **Scaling nodes**: Cardinality is the sum of a constant value from a given distribution multiplied by the scaling factor.
- **Growing nodes**: Cardinality is a random neighbour's cardinality multiplied by a constant value from a given distribution.

In figures A.2 and A.3, we visualise an example of a possible schema with its initial cardinality for two scaling factors SF. The fixed green nodes show that their cardinality does not increase with a greater SF. The growing and scaling nodes grow with SF. However, the growing nodes grow indirectly since they are a sum of the scaling node C cardinality.

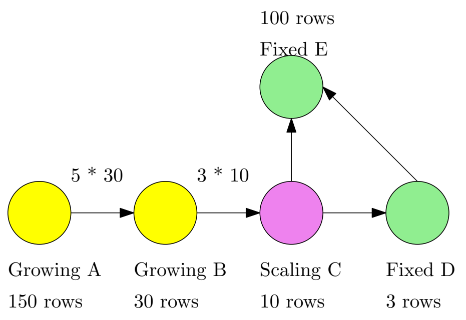

Figure A.2: Initial cardinality with $SF = 1$

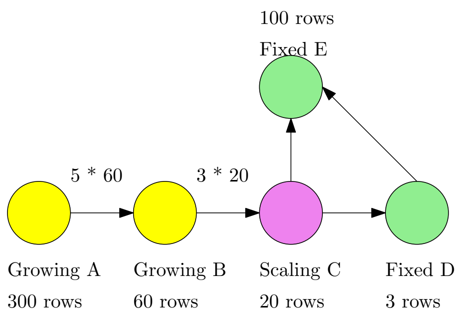

Figure A.3: Initial cardinality with $SF = 2$

A.5 Query Simulation

A.5.1 Modelling

The query workloads are the core of our enhancement of the original ℓ-BCNF algorithm. The original algorithm defines the level ℓ of maintenance and the joining efficiency of the tables in the schema. Therefore, we only need to capture the properties of the queries that affect the update and joins. Specifically, which tables does a query join and how many updates occur on tables. Most modern RDBMS track the history of queries in the form of a query store where we can extract occurrences of queries.

The properties of the queries that we want to capture are:

- Text: The query that was executed
- Type: Read or update
- Sources: The nodes in the graph that selected or updated
- Join: The edges from the graph which were used to join the query
- Invocation: The number of calls
- Cost: The mean number of affected rows

With these properties we are able to estimate what the effect is on query when we normalise or denormalise tables.

A.5.2 Generation

For testing purposes, we also need to generate a workload to show how algorithms will behave in different scenarios. Similarly to the schema simulation, the goal of this simulation is to generate realistic workloads. We have split the workloads into CRUD (Create, Read, Update, and Delete).

In this section we discuss how each of these queries is generated, and how the cost and number of invocation is determent.

First, create queries are generated. One create query is generated for each of the scaling and growing tables, since fixed tables should stay constant as of their definition. The number of invocations of these create queries is bound to the fraction of the initial cardinality of all tables, and a growing factor that indicates how many create queries need to be generated. For example, if we have a set of nodes V where the initial cardinality of $v \in V$ is $|v|$ and we want to insert n rows, then the number of create queries is the fraction of the total initial cardinalities multiplied by n. Equation A.2 shows this function.

$$cost(v) = 1 \tag{A.1}$$

$$inv(v, n) = \frac{n \cdot |v|}{\sum_{v_i \in V} |v_i|} \tag{A.2}$$

The second type of queries that are generated are the delete queries. Similarly, to create queries, do we want to keep the same balance between table cardinalities as defined in the initial cardinality. However, delete queries add extra complexity, since cascading behaviour means that a delete on a single table often triggers multiple deletions on other tables. Our attempt to mimic this behaviour correctly is to use the multipliers that are used to set the initial cardinality of growing tables. When the cardinality of table b is defined by the cardinality of table a times x, then $x = \frac{a}{b}$. If we then denote the set of tables that got their cardinality from table a as children[a]. We can calculate the cost of deleting a row from a as Equation A.3. The number of invocations is an equal distribution over all queries, since the cascading effect already balances cardinalities. This function can be seen in Equation A.4.

$$cost(v) = \begin{cases} 1 & \text{if v has no children} \\ \sum_{u \in \text{children}[v]} \frac{|u| \cdot cost(u)}{|v|} & \text{otherwise} \end{cases} \tag{A.3}$$

$$inv(v, n) = \frac{n}{\sum_{v_i \in V} |v_i|} \tag{A.4}$$

Generating update queries does not follow a pattern like create and delete queries. Therefore, we choose to create these queries at random. They are created by randomly selecting a number of tables from selected table categories. Each of them gets assigned a random cost and invocations from a given distribution.

Finally, read queries are generated. Read queries are the most complex to simulate, since they have the highest variation in complexity. To address the levels of complexity, we split the read queries into two types, simple and complex queries. For both types, we define the number of queries to generate, and two ranges form which to sample the depth and degree of the joins to make for a query. The depth indicates the maximum joins in a row (e.g. $A \to B \to C$), and the degree the number of joins on a single node (e.g. $A \to B, A \to C$). Invocations for read queries are controlled by a structure similar to that defined in the LDBC benchmark [3]. We create a sequence of simple queries to execute on for each complex query. Simple queries are selected by visiting the connected nodes from the source nodes of complex queries. The invocations are then determined by giving each read query a probability of being invoked. When a complex query is invoked, its query sequence is then also invoked. Sequences cause simple queries to be more common than complex ones if they exist, and they simulate the behaviour of selecting extra information from a result. For example, when a user's identifier is returned from a complex query, a simple query might show that user's friends.