Query Solvers on Database Covers

Wouter Verlaek
Kellogg College
University of Oxford

Supervised by Prof. Dan Olteanu

A thesis submitted for the degree of
Master of Science in Computer Science
Trinity 2018

Word count: 20268

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. Dan Olteanu for his continuous support throughout the project, the many productive meetings, and the great feedback he has given during the last five months.

Additionally, I would like to thank Postdoc Ahmet Kara for his support and helpful contributions to the project, both during and outside of the many meetings.

Finally, I also want to thank my family for their love and support during the past year.

Abstract

This thesis investigates the problem of solving aggregate queries over the natural join of relations, where the size of the join result can be exponentially larger than the sizes of the individual relations joined together. A recently introduced relational representation called a cover is a succinct lossless representation of the flat result of a natural join. This thesis gives new algorithms to solve aggregate queries over covers, details on trade-offs encountered while implementing them, and describes an extensive experimentation benchmarking the algorithms on several large datasets. We show that our algorithms are outperforming existing solutions by at least 37% on a subclass of queries called hierarchical join queries.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this thesis, we exploit and push further very recent development on lossless, succinct representation of relational data. The result of a join entails large amounts of redundancy in both its representation as well as its computation. The problem of solving aggregate queries over large join results lies at the foundation of relational databases and has therefore received tremendous attention over five decades, since the seminal paper by Ted Codd that introduced the relational model [3]. Several approaches have been developed over the years that attempt to solve this problem, such as factorised databases [4]. This factorised representation implements structure-based compression through a graph-like structure that is compact and lossless, and has been shown to support select-project-join queries [2] as well as aggregation and ordering [1]. However, its "complex" structure makes it hard to process queries in-memory, as it suffers from a significant amount of cache misses. Alternative approaches include columnar stores such as MonetDB [5] and C-Store [11], which implement value-based compression techniques such as run-length encoding to reduce the data size. Whereas structure-based compression may lead to asymptotically smaller representations, this is not the case for value-based compression. Recently, covers [6] have been proposed as a new compressed lossless relational representation for relational data. It combines the best of both relational and factorisation worlds while being as succinct as graph-based factorised databases. It remains relational and can be accommodated by virtually all relational data stores. As covers can be processed tuple by tuple, they do not have the same structural complexity problems as the factorised representation. This gives the conjecture that covers have less cache misses and better code quality than factorised representations.

Factorised databases [2] have been used extensively for in-database machine learning [7, 9, 12], where aggregate queries play an important role. Due to the complex structure of factorised representations, it would be interesting to develop aggregate

query solvers that operate over covers instead, taking advantage of their relational structure while still being as succinct as the factorised representation. So far, no approach exists yet that leverages the power and elegance of covers for query processing.

Another existing approach that attempts to improve the performance of solving aggregates over join queries is eager aggregation [13], which pushes the aggregate past the join such that the computation is no longer dependent on the possibly exponential join result. We show how this approach compares against solving aggregate queries over covers.

1.1 Contributions

We introduce several approaches for solving aggregate queries over covers. These covers may represent entire relations or succinct representations of results of join queries.

Our main approach HierarchicalSolver relies on a novel sorting technique called tree-sorting. In contrast to standard sorting, that sorts a relation according to a total order of (some of) its attributes, tree-sorting does not assume a total order on these attributes. Tree-sorting may sort different partitions of a cover over different attributes. This exploits the unique properties of the input cover so that the sorted cover still represents the same relation as the original cover.

Tree-sorting fits naturally with computing aggregates in one pass over covers that have a hierarchical schema, which are covers of hierarchical joins. In case of covers with a non-hierarchical schema, we introduce a novel decomposition scheme for arbitrary schemas into a graph of hierarchical sub-schemas. For each hierarchical sub-schema, we can deploy HierarchicalSolver in one pass and construct a temporary result, which is fit as input to the HierarchicalSolver in subsequent passes. We conjecture that the hierarchical schemas are the only schemas that allow for computing aggregates in one pass and without additional memory.

We then introduce a refinement of HierarchicalSolver that can parallelise the computation of partial aggregates for hierarchical sub-schemas.

Additionally, we implement two variants of eager aggregation, where one operates over the projections of a cover and the other over the separate relations in the join queries used in the experiments. These serve as competitors for HierarchicalSolver, as they do not operate over the full join result.

This suite of aggregate solvers has been implemented in Kotlin[1], a language that runs on the Java virtual machine. We benchmarked them on several datasets. Our experimental findings show that HierarchicalSolver outperforms the next best competing solver by up to 37% on hierarchical join queries, while giving up its lead on non-hierarchical datasets where it is heavily impacted by sorting the multiple sub-schemas. We do show possibilities of increasing the performance of HierarchicalSolver on non-hierarchical join queries, such as smarter tree-sorting, where one can skip sorting certain parts of the sorting tree after having sorted the subtree for previous sub-schemas.

1.2 Outline

- **Chapter 2** introduces covers.
- **Chapter 3** proposes a new sorting technique using the structure of a cover and shows how we can solve aggregates over hierarchical join queries.
- **Chapter 4** extends our method of solving aggregate queries over covers to arbitrary acyclic join queries.
- **Chapter 5** describes the implementation of the hierarchical solver, as well as the competing solvers used in the benchmarks.
- **Chapter 6** contains an experimental evaluation of our proposed methods.
- **Chapter 7** is aimed at giving a conclusive overview of our contributions, and provides possible directions for future work.
- **Appendix A** describes the datasets used in greater detail.
- **Appendix B** specifies our experiment setup.
- **Appendix C** includes useful tools that were developed during the project.

[1] `https://kotlinlang.org/`

Chapter 2

Covers: Succinct and Lossless Representations of Join Results

This chapter introduces covers, which are succinct lossless relational representations of query results [6]. The number of tuples in a cover can be much less than the size of the relation it represents. There exist relations R with n attributes which have covers of size $\sqrt[n]{|R|}$, where $|R|$ is the number of tuples in R. The gap in size allows for solving queries more efficiently. Instead of computing the result of a query over a relation R, one can compute it over a cover of R. An additional benefit of covers is the increase of data locality. All projections over the bags in the decomposition of a cover are in the same relation, and scanning over these projections can be simulated by a single scan over the cover relation. When the cover represents the result of a join query, this can be more efficient than scanning the individual relations in the join separately. However, not all relations have covers that are of smaller size. We will see covers of equal size later on in the experiments. Since covers are a subset of the relation they represent, no cover is larger than the relation it represents.

First we introduce several important notions. A schema S is a set of attributes. A tuple over a schema S is a mapping from the attributes in S to data values. A relation R over the schema S is a finite set of tuples over S. Let $S(R)$ denote the schema of relation R. Next, we give a brief introduction to covers. A more detailed description can be found in [6].

Definition 2.1. *A* decomposition C *of a schema* S *is a set of non-empty subsets* $S_1, \ldots, S_k$ *(called bags) of* S *such that the union of these subsets is* S*, i.e.,* $\forall i \in [k] : \emptyset \subset S_i \subseteq S$ *and* $S = S_1 \cup \cdots \cup S_k$*.*

A decomposition C *is* acyclic *if it can be represented by a tree* T *as follows:*

- *Coverage: The nodes in T are the schemas in C.*
- *Connectivity: For any two nodes S_i and S_j, the attributes in their intersection must appear in all nodes along the path between S_i and S_j in T.*

Definition 2.2. *A* cover *of a relation R is a pair (K, C), where K is a subset of R and C is a decomposition $\{C_1, \ldots, C_n\}$ of the schema of R, that satisfies the following two properties:*

- *Recoverability: R can be recovered exactly from K by taking the natural join of the projections of K onto the schemas $C_1, \ldots, C_n$: $\pi_{C_1}K \bowtie \cdots \bowtie \pi_{C_n}K = R$.*
- *Minimality: There is no strict subset of K that satisfies the recoverability property.*

Covers can be used to represent the result of a natural join query, where the decomposition of the cover is the set of schemas of the relations in the join query. By construction, the decomposition of a cover is always acyclic, even for arbitrary join queries [6]. For non-acyclic queries, covers are constructed by first reducing the query to an acyclic query.

Example 2.3. Consider the following relation R over the schema $\{A, B, C\}$.

R		
A	B	C
1	2	1
1	2	2
1	2	3
2	2	1
2	2	2
2	2	3
3	1	3
4	1	3

K_1		
A	B	C
1	2	1
2	2	2
2	2	3
3	1	3
4	1	3

K_2		
A	B	C
1	2	3
1	2	2
2	2	1
4	1	3
3	1	3

N_1		
A	B	C
1	2	1
2	2	3
3	1	3
4	1	3

N_2		
A	B	C
1	2	3
1	2	2
2	2	2
2	2	3
3	1	3
4	1	3

Let us consider the decomposition $C_1 = \{\{A, B\}, \{B, C\}\}$ of the schema $\{A, B, C\}$. This decomposition is acyclic: it can be represented by a tree with two connected nodes $\{A, B\}$ and $\{B, C\}$. Relations K_1 and K_2 are covers of relation R with decomposition C_1. Joining the projections of the bags in C_1 over K_1 and K_2 results in relation R: $\pi_{A,B}K_1 = \pi_{A,B}K_2 = \pi_{A,B}R = \{(1, 2), (2, 2), (3, 1), (4, 1)\}$ and $\pi_{B,C}K_1 = \pi_{B,C}K_2 = \pi_{B,C}R = \{(2, 1), (2, 2), (2, 3), (1, 3)\}$.

In contrast, relations N_1 and N_2 together with decomposition C_1 are not covers of R. Relation N_1 violates the Recoverability property: $\pi_{A,B}N_2 \bowtie \pi_{B,C}N_2 \subset R$,

it cannot reconstruct tuple $(2, 2, 2)$ for instance. Relation N_2 does not satisfy the Minimality property: it strictly contains cover relation K_1.

The decomposition $C_2 = \{\{A\}, \{B\}, \{C\}\}$ is acyclic, since it can be represented by any tree with the three nodes $\{A\}$, $\{B\}$, and $\{C\}$. However, R does not admit a cover with decomposition C_2, as there is no relation such that the Cartesian product of its projections on each of the attributes A, B, and C is equal to R.

The only cover of R with the trivial decomposition $C_3 = \{\{A, B, C\}\}$ is (R, C_3). This decomposition is acyclic as it is represented by the tree with one node $\{A, B, C\}$.

Chapter 3

Aggregates over Covers with Hierarchical Schemas

This chapter introduces a special class of schemas for covers called hierarchical schemas, where the schema of a cover refers to the set of bags in the decomposition of the cover. We describe a unique sorting technique for covers, and show how to use this to solve aggregate queries over covers with hierarchical schemas in only a single pass over the cover and no additional memory.

3.1 Definitions

We introduce the definitions related to hierarchical schemas.

Definition 3.1. *A* hierarchical cover schema H *(hierarchical schema for short) satisfies the following property, where* X *and* Y *are attributes in* H *and* $bags(X)$ *is a function mapping each attribute* X *to the set of bags in* H *containing* X*:*

$$\begin{aligned} \text{For all } X, Y \text{ in } H: \quad & bags(X) \cap bags(Y) = \emptyset \\ & \text{or } bags(X) \subseteq bags(Y) \\ & \text{or } bags(Y) \subseteq bags(X) \end{aligned}$$

Example 3.2. Consider the decomposition $C_1 = \{\{A, B, C\}, \{A, B, D\}, \{A, E\}\}$. This decomposition satisfies the property in Definition 3.1, as for any pair of attributes X, Y in C_1 either $bags(X) \subseteq bags(Y)$, $bags(Y) \subseteq bags(X)$, or $bags(X) \cap bags(Y)$ is empty. The decomposition is therefore a hierarchical schema.

Now consider adding the bag $\{E, F\}$ to C_1, resulting in $C_2 = \{\{A, B, C\}, \{A, B, D\}, \{A, E\}, \{E, F\}\}$. This new decomposition is not hierarchical, since the pair of attributes A, E does not satisfy the property in Definition 3.1:

$bags(A) = \{\{A, B, C\}, \{A, B, D\}, \{A, E\}\}$ is neither a subset of, nor contains $bags(E) = \{\{A, E\}, \{E, F\}\}$, and $bags(A) \cap bags(E) \neq \emptyset$.

Following [8], where variable orders are called d-trees:

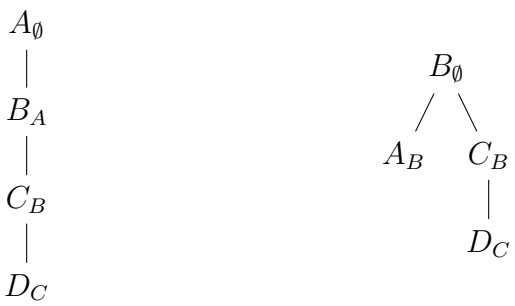
Definition 3.3. *A* variable order Δ *for a decomposition C is a partial ordering of the attributes in C, such that for each bag in C its attributes are along the same root-to-leaf path. Formally, a* variable order Δ *is a pair (F, key), where*

- *F is a rooted forest with one node per attribute in C*
- *key is a function mapping each attribute X to a subset of its ancestor attributes in F.*

A variable order Δ over decomposition C has the following properties:

- *For each bag in C, its attributes lie along the same root-to-leaf path in F. For any such attributes X and Y, $X \in key(Y)$ if X is an ancestor of Y.*
- *For every child Y of X, $key(Y) \subseteq key(X) \cup \{X\}$.*

Example 3.4. Consider the decomposition $C = \{\{A, B\}, \{B, C\}, \{C, D\}\}$. Two possible variable orders for C are:



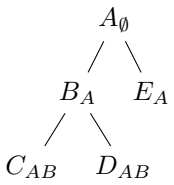
For each node, the attributes in its key are indicated in subscript.

Definition 3.5. *A* hierarchical variable order Δ_H *is a subclass of variable orders, which satisfy the following property:*

- *For any attribute X, $key(X)$ contains attribute Y if and only if X and Y are in the same bag and Y is an ancestor of X.*

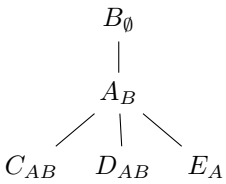
This additional property is equivalent to saying that the attributes of every bag in the decomposition must be on an uninterrupted root-to-leaf path in Δ_H, with no other attributes in between on that path.

Example 3.6. Consider the hierarchical schema C_1 in Example 3.2 again. A hierarchical variable order Δ_H for C_1 would be the following:



In fact, this is the only hierarchical variable order for C_1, no other variable order satisfies the hierarchical property for this decomposition. We can see that for all bags $\{A, B, C\}$, $\{A, B, D\}$, and $\{A, E\}$ in C_1 its attributes lie on a root-to-leaf path.

The following is a valid variable order for C_1, but is not hierarchical:



The key of E does not contain all of E's ancestors: it is missing B.

For ease of notation, we will discard the keys for nodes in a hierarchical variable order, as they can be inferred from the ancestors of each node.

Proposition 3.7. *Every hierarchical schema has at least one hierarchical variable order.*

Proof. Given a hierarchical schema H, a hierarchical variable order Δ_H for H can be constructed as follows: Let $count(X)$ be a function mapping each attribute X to the number of bags in H containing X, i.e., $count(X) = |bags(X)|$. Order the attributes in H on the *count* function in descending order, such that the attribute occurring in the most bags comes first. Mark the parent of each attribute as null. Then, for each attribute X in descending order of *count*:

- Add X to Δ_H under its marked parent, adding X as a new root if its parent is null.
- For each bag $S \in bags(X)$, mark X as the parent of each attribute Y in S.
- Continue with the next attribute.

In this construction, every attribute X is placed under an attribute Y where X and Y are both in the same bag of H. Additionally, since H is hierarchical, no

Add A as new root, and set A as parent for B, C, D, and E:

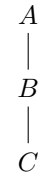
X	A	B	C	D	E
$count(X)$	3	2	1	1	1
$parent$	-	A	A	A	A

B has A marked as its parent, so add B under A.
Attributes A, C, and D are in bags with B, mark B as parent for these:

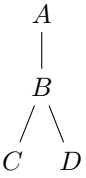
X	A	B	C	D	E
$count(X)$	3	2	1	1	1
$parent$	-	A	B	B	A

Remaining attributes C, D, and E do not mark parents of yet unplaced nodes.
Finish by adding them under their marked parents:

X	A	B	C	D	E
$count(X)$	3	2	1	1	1
$parent$	-	A	B	B	A



X	A	B	C	D	E
$count(X)$	3	2	1	1	1
$parent$	-	A	B	B	A



X	A	B	C	D	E
$count(X)$	3	2	1	1	1
$parent$	-	A	B	B	A

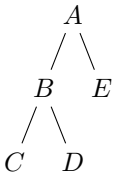


Figure 3.1: Construction of Δ_H for C_1 in Example 3.2.

attribute X will have an ancestor Z such that there is no bag in H containing both X and Z. We will show the latter by contradiction.

Assuming such a Z does exist, then $bags(Z)$ neither contains, nor is a subset of $bags(X)$. In the case $bags(Z) \cap bags(X) = \emptyset$, then there must be another attribute Y on the path between Z and X for which $bags(X) \cap bags(Y) \neq \emptyset$ but neither of them being a subset of the other. This would cause a contradiction, as H would not be hierarchical then. Therefore, all ancestors of X are in a bag of H with X, satisfying the property of a hierarchical variable order. □

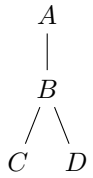
Example 3.8. The steps to construct Δ_H for hierarchical schema C_1 in Example 3.2 are shown in Figure 3.1.

Proposition 3.9. *If a decomposition admits a hierarchical variable order then the decomposition must be hierarchical.*

Proof. This follows from each root-to-leaf path in the hierarchical variable order Δ_H of decomposition C representing a bag in C. For any two attributes X, Y, either they occur in separate subtrees in Δ_H, or one is an ancestor of the other. In the case of the latter, assuming without loss of generality that X is an ancestor of Y, it holds that $bags(Y) \subseteq bags(X)$. All bags are along a root-to-leaf path in Δ_H, therefore, any bags containing Y will also have to contain X, since X is on the path from Y to the root. When X and Y are separated in different subtrees, $bags(X) \cap bags(Y) = \emptyset$, again because bags are only along root-to-leaf paths. A third case is that $X = Y$, but then trivially $bags(X) \subseteq bags(Y)$. Therefore, if a decomposition admits a hierarchical variable order, the decomposition must be hierarchical itself. □

Definition 3.10. *The* root attributes *of a tree are the attributes along the path from the root to the first branch in the tree.*

Example 3.11. The root attributes for the tree in Δ_H of C_1 in Example 3.2 is the singleton set $\{A\}$. Removing bag $\{A, E\}$ from C_1 to get C_2, a hierarchical variable order for C_2 might be the following, with $\{A, B\}$ as its root attributes:



3.2 Tree-Sorting: A Novel Sorting Technique for Covers

Instead of solving an aggregate query over a relation R, we would like to solve it over the cover of R instead, where the cover has decomposition C. Since the join of the projections of each bag in C over the cover results in relation R, we can solve an aggregate over the natural join over these projections of the cover instead. To solve such an aggregate query efficiently, we would like to be able to detect and aggregate the unique tuples of each bag in the decomposition using a single scan over the cover. In order to detect unique tuples while using only a constant amount of additional memory, we require that the tuples for each bag appear in sorted order in the cover when taking the projection of that bag.

Naively, one could sort the cover relation for each bag repeatedly. Instead, we propose a novel sorting algorithm, called tree-sorting, which sorts a cover with a hierarchical schema once such that it is sorted over all bags in its hierarchical schema simultaneously, in the same linearithmic running time as standard sorting. Tree-sorting takes advantage of the unique structure of a cover, allowing it to do more complex reordering of data values in the cover compared to regular sorting over a standard relation.

The tree-sorting algorithm accepts a cover with decomposition C and a variable order Δ over C. Remember that for each bag in C its attributes are on the same root-to-leaf path in Δ. Note that hierarchical variable orders are a subclass of variable orders and therefore are valid input. When performed, tree-sorting results in a cover of the same size as the original cover, representing the same relation, while being sorted on each root-to-leaf path in Δ. Sorting over a hierarchical variable order therefore results in the projections of each bag being sorted.

Example 3.12. Considering the hierarchical schema C_1 in Example 3.2 again, sorting the cover on its hierarchical variable order will result in the projections $\{A, B, C\}$, $\{A, B, D\}$, and $\{A, E\}$ over the cover being sorted.

Calling TreeSort in Algorithm 3.1 on each root attribute of a variable order Δ sorts the cover over Δ. The attribute sets the trees in Δ are disjoint, so each tree operates over a non-overlapping set of columns in the cover. Focusing on a single tree T we start by sorting the entire range of tuples in K on the root attribute A of T, only reordering the values in tuples for attributes in T. Next, for each block of distinct values for A, we recurse to the children of A in T. For each child attribute C of A, call TreeSort for the block of tuples with equal value for A. Since the attributes in the subtree of each child are disjoint, these recursive calls again operate over their own partition of the cover.

When sorting on an attribute X over a range of tuples $[from, to)$, the invariant is that for all ancestors of X in Δ their values do not change over the given range. Since only values for attributes in the same relations as X are moved, the projections of each bag are kept intact, only reordering the tuples for a bag. The sorted cover is therefore equivalent to the original cover, representing the same relation.

Figure 3.2 shows each step in the tree-sorting algorithm for an example cover of hierarchical schema C_1 in Example 3.2 that has hierarchical variable order Δ_H. First, we sort the entire cover over root attribute A of Δ_H, moving all attributes in tuples as we sort. Next, we group the tuples by distinct values of A. There are two blocks:

Algorithm 3.1 Tree-sort for node in variable order Δ

```
1: procedure TreeSort(K, A, from, to)
2:     ▷ values for ancestors of A in Δ are equal in range [from, to)
3:     attrsToSort ← attributes in subtree of Δ rooted at A
4:     sort block with rows [from, to) and columns attrsToSort on attribute A

5:     for each block [blockFrom, blockTo) of distinct values for A in [from, to) do
6:         for each child C of A in Δ do
7:             TreeSort(K, C, blockFrom, blockTo)
```

rows 1 and 2 are for $A = 1$ and rows 3 - 6 for $A = 2$. For each block, we split into separate calls for each child of A in Δ_H, sorting disjoint regions of the cover on these child attributes B and E. Since E is a leaf in Δ_H, only values for E are sorted, while leaving values for other attributes in place. On the other hand, B still has the two child attributes C and D in Δ_H. Therefore, when we sort on B we also reorder the values for attributes C and D. After having completed the sorting for B, we again find distinct blocks of values for B (per distinct block of A values, effectively splitting into blocks of distinct values for $[A, B]$). Then finally for each block of distinct $[A, B]$ values, the algorithm recurses and sorts the values for C and D independently. The result is shown in Figure 3.2 as well, and Figure 3.3 shows the projections of the bags in C_1 over the original cover as well as over the sorted cover. The projections over the sorted cover are sorted themselves as well, while containing exactly the same tuples as in the original projections but reordered.

3.2.1 Group Attributes in Paths

As we will see in the datasets used in the experiments, many datasets contain relations in the natural join that have a significant number of attributes unique to that relation. This results in the hierarchical variable order for the cover of the join containing many long chains of attributes. For example, the hierarchical variable order for the cover of the Housing datasets, as seen in Appendix A, contains one root attribute with six chains of child attributes, each containing between 2 and 10 attributes. Calling the recursive TreeSort algorithm for each node in these chains individually significantly impacts the performance, whereas it is possible to group chains together and execute the algorithm on each "path" of attributes in the hierarchical variable order. The Housing dataset will then need a call on the root attribute and on each of its six child paths, essentially reducing the number of steps to process in the hierarchical variable

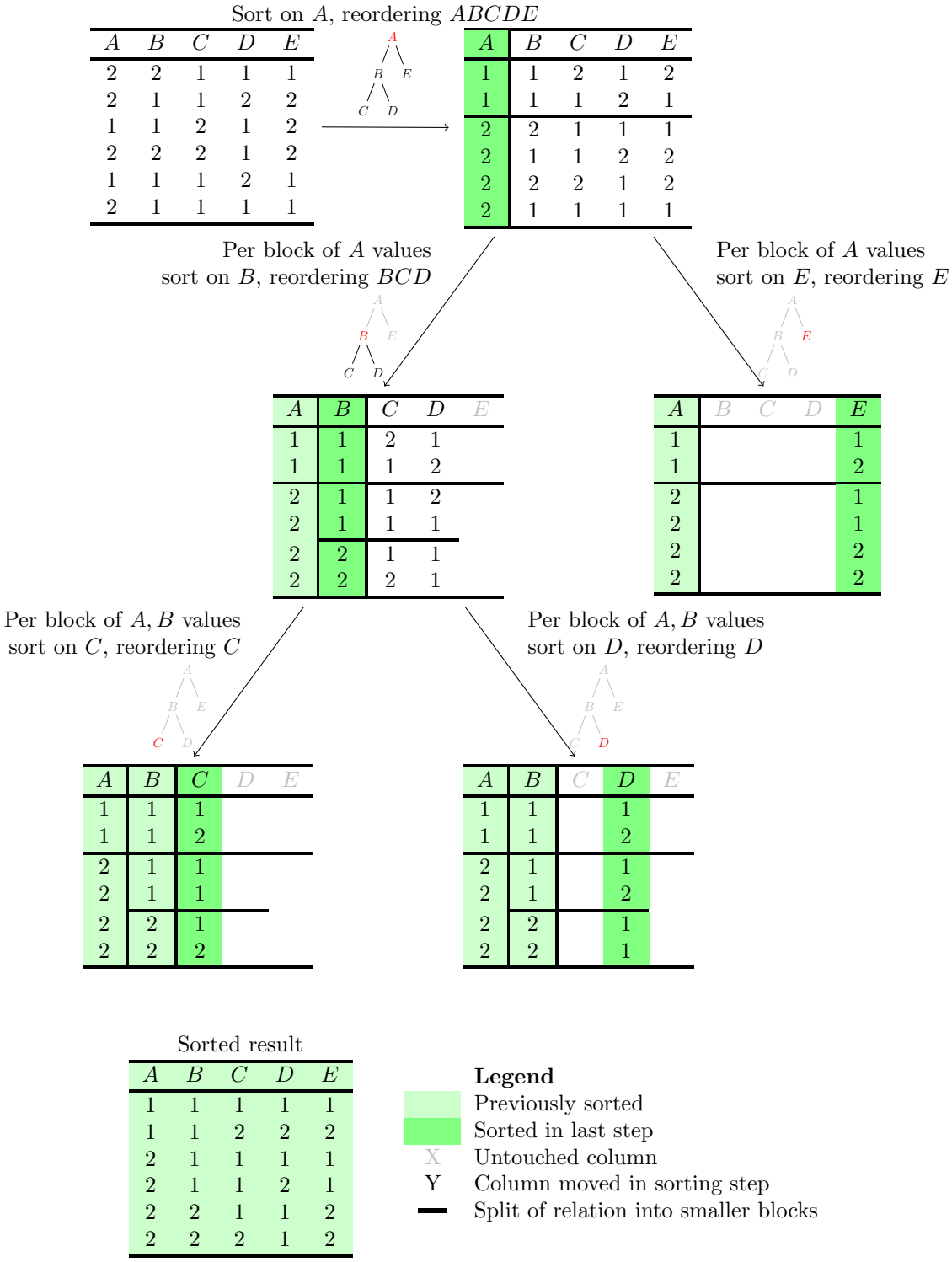


Figure 3.2: Visualisation of steps in tree-sorting algorithm.

A	B	C
2	2	1
2	1	1
1	1	2
2	2	2
1	1	1
2	1	1

A	B	D
2	2	1
2	1	2
1	1	1
2	2	1
1	1	2
2	1	1

A	E
2	1
2	2
1	2
2	2
1	1
2	1

A	B	C
1	1	1
1	1	2
2	1	1
2	1	1
2	2	1
2	2	2

A	B	D
1	1	1
1	1	2
2	1	1
2	1	2
2	2	1
2	2	1

A	E
1	1
1	2
2	1
2	1
2	2
2	2

Figure 3.3: Projections of bags in C_1 over original unsorted cover (left) and sorted cover (right).

order from 27 to 7; a decrease of almost 75%. Adapting Algorithm 3.1 to operate over paths of attributes results in Algorithm 3.2.

Algorithm 3.2 Tree-sort for path in variable order Δ

```
1: ▷ Sort cover K for subtree in Δ rooted at path A_p in range [from, to)
2: ▷ Initial call: TreeSortPath(K, root attributes of Δ, 0, K.size)
3: procedure TreeSortPath(K, A_p, from, to)
4:     ▷ values for ancestors of A_p in Δ are equal in range [from, to)
5:     ▷ sort on A_p
6:     attrsToSort ← attributes in subtree of Δ rooted at first attribute of A_p
7:     sort block with rows [from, to) and columns attrsToSort on attributes A_p
8:     for each block [blockFrom, blockTo) of distinct values for A_p in [from, to) do
9:         for each child path C_p of A_p in Δ do
10:            TreeSortPath(K, C_p, blockFrom, blockTo)
```

3.2.2 Memory and Running Time Analysis

The analysis of memory usage and running time of the tree-sorting algorithm depends largely on the underlying "regular" sorting algorithm used in each of the recursive steps. When the cover to sort has m attributes, tree-sorting will perform at most m regular sorts over the cover, or less if attributes are grouped together. Each of these m regular sorts operates over a smaller number of columns the deeper you get in the hierarchical variable order, as well as the tuples in these columns being recursively divided into smaller blocks for distinct ancestor values. Both reduce the time it takes to sort. Additional to the memory used for the regular sorting, tree-sorting uses only $O(m)$ memory for storing the attributes to sort on per step, as well as the stack being increased by the recursive calls (up to the maximum depth of the hierarchical variable order).

Therefore, assuming m is considered constant in terms of the running time and memory usage, the total duration and memory usage to sort a cover with n tuples is asymptotically the same as the duration and memory usage of the chosen regular sorting algorithm. If the regular sorting algorithm runs in linearithmic time, so will the tree-sorting algorithm.

In the case of Δ being just a path, effectively sorting similarly to a regular sorting algorithm over a list of attributes, tree-sorting will group that path of attributes together and perform a single regular sort over that path. Therefore, the behaviour of tree-sorting gracefully falls back to regular sorting when sorting over a list of attributes.

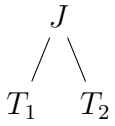
3.2.3 Cover-Join Operator and Resulting Sort Tree

To construct a cover for relation R from a set of projections over R, a cover-join operator is introduced in [6]. We will show how to implement this cover-join operator such that the covers are sorted on a particular sorting tree. Following the definition from [6]:

Definition 3.13. *The* cover-join *of two relations R_1 and R_2, denoted by $R_1 \mathbin{\mathring{\bowtie}} R_2$, computes a cover of their join result over the decomposition with bags $S(R_1)$ and $S(R_2)$.*

The suggested way in [6] to implement this cover-join algorithm such that a cover of minimum size is constructed is as follows: for each join key, get the complete bipartite graph between tuples in either relation for that join key. Let U and V be the lists of nodes in this bipartite graph, such that $|U| \leq |V|$. Let the minimum edge cover be the list of $|V|$ edges, such that edge i connects node $min(i, |U|)$ in U with node i in V. For each edge i in the edge cover, adding the tuple represented by that edge to the cover-join result in ascending order of i preserves any sort order that was present in R_1 and R_2 for this particular join key, since the edges in the edge cover connect tuples in R_1 and R_2 in the same relative order as these tuples appear in R_1 and R_2. Add the tuples for each bipartite graph in increasing order of join keys to the cover-join result. The resulting cover is sorted on the join attributes, and for each join key, the cover is sorted on the original sort orders in R_1 and R_2.

Let T_1 and T_2 be the trees R_1 and R_2 are sorted on, respectively. These can be empty if the relations were not sorted. After applying the cover-join $R_1 \mathbin{\mathring{\bowtie}} R_2$, joining on attributes $J = S(R_1) \cap S(R_2)$, the resulting cover is sorted on tree:



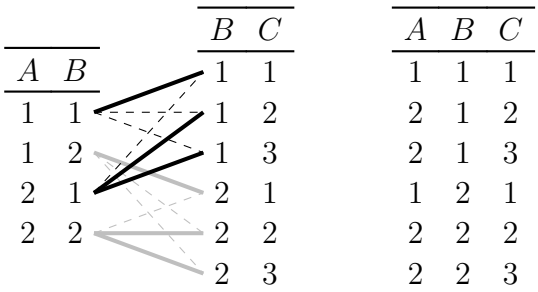
Example 3.14. Consider the following (sorted) tables R_1, R_2, and R_3:

R_1	
A	B
1	1
1	2
2	1
2	2

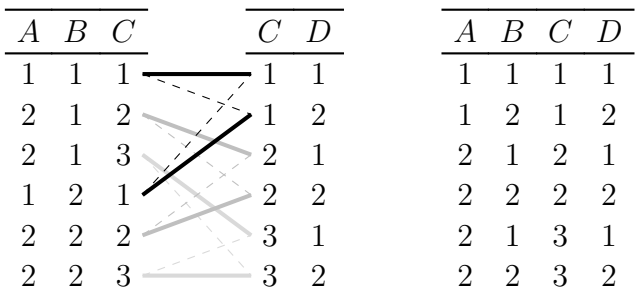
R_2	
B	C
1	1
1	2
1	3
2	1
2	2
2	3

R_3	
C	D
1	1
1	2
2	1
2	2
3	1
3	2

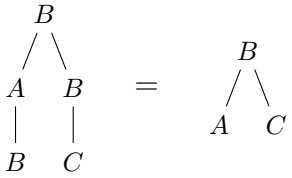
Let $(R_1 \overset{\circ}{\bowtie} R_2) \overset{\circ}{\bowtie} R_3$ be the cover-join plan. The first step $R_1 \overset{\circ}{\bowtie} R_2$ is as follows, with on the left the cover-join between R_1 and R_2, showing the complete bipartite graphs for each join value, and selected minimum edge cover as bold lines. On the right is the result of the cover-join.



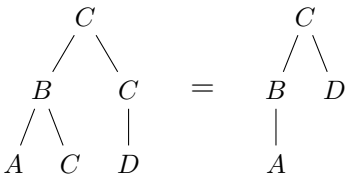
Next, applying the cover-join between the result of $R_1 \overset{\circ}{\bowtie} R_2$ and relation R_3 gives the following:



Initially, R_1, R_2, and R_3 are sorted on paths $A-B$, $B-C$, and $C-D$ respectively. Therefore, $R_1 \overset{\circ}{\bowtie} R_2$, where $J = \{B\}$, is sorted on:



Finally, $(R_1 \mathring{\bowtie} R_2) \mathring{\bowtie} R_3$, where $J = \{C\}$, is sorted on:



Cover-Join and Hierarchical Queries For hierarchical queries, every cover-join step will be joining on at least the root attributes of the hierarchical query, as every bag in a hierarchical query contains these attributes. Therefore, a cover constructed using the described cover-join algorithm is always sorted on at least its root attributes.

3.3 Aggregate Solver

In this section we show how to build upon the tree-sorting algorithm and create an efficient aggregate query solver over covers with hierarchical schemas that solves the aggregate query in one pass over the cover using only a constant amount of memory.

We conjecture that hierarchical schemas are the only schemas that allow for computing aggregates in one pass and without additional memory.

3.3.1 Aggregate over Tree-Sorted Cover

Once a cover relation K of a hierarchical schema H is sorted over its hierarchical variable order Δ_H, we can easily compute different types of aggregate queries in a single scan over K while using $O(1)$ additional memory. Having sorted the cover on Δ_H, for each bag S in H the projection of S onto K will be sorted on the root-to-leaf path of S in Δ_H. This allows for the aggregate to be accumulated in one scan. We will consider count and sum aggregate queries here.

Algorithm 3.3 shows how to compute an aggregate over a cover with hierarchical schema sorted on Δ_H. Starting at the path of root nodes of Δ_H, split the cover into blocks of distinct values for the root attributes. For each block, recurse for every child path of attributes in Δ_H, computing their aggregate over the block, which is then multiplied with the aggregates of the other children (representing the cross-product

in the join between tuples with the same join-key: the current ancestor values). The result of this is the aggregate for the current block of distinct root values. Summing these block aggregates, where each block is the aggregate for a particular join key, results in the total aggregate, which is then returned.

Additionally, one should check if the current path of attributes contains the aggregate attribute in case the aggregate query is a sum over an attribute. If so, multiply the collected aggregate from the children with the value for the aggregate attribute in the current block before adding it to the aggregate sum to be returned.

We assumed that the hierarchical variable order consists of a single tree. If instead the variable order is a forest, representing a decomposition with disjoint subsets of bags, one should compute the aggregate for each tree separately, and multiply the results of each tree, corresponding to a cross product between the disjoint groups of bags.

Algorithm 3.3 Compute aggregate for path in Δ_H over sorted cover on Δ_H

```
1: ▷ Compute aggregate for path A_p in Δ_H in range [from, to) over cover K
2: ▷ Initial call: AGGREGATE(K, root attributes of Δ_H, 0, K.size, aggAtr)
3: procedure AGGREGATE(K, A_p, from, to, aggAttr)
4:     ▷ values for ancestors of A_p in Δ_H are equal in range [from, to)
5:     ▷ values for A_p are in sorted order in [from, to)
6:     aggregate ← 0
7:     for each block [blockFrom, blockTo) of distinct values for A_p in [from, to) do
8:         ▷ every tuple in [blockFrom, blockTo) has same values for A_p
9:         if aggAttr in A_p then
10:            ▷ get value of aggregate attribute in current block
11:            ▷ use tuple at index blockFrom
12:            blockAggregate ← π_aggAttr K[blockFrom]
13:        else
14:            blockAggregate ← 1
15:        for each child path C_p of A_p in Δ_H do
16:            childAggregate ← AGGREGATE(K, C_p, blockFrom, blockTo, aggAttr)
17:            blockAggregate ← blockAggregate * childAggregate
18:        ▷ computed aggregate for block, add to total aggregate
19:        aggregate ← aggregate + blockAggregate
20:    return aggregate
```

3.3.2 Merging Tree-Sort and Aggregate Steps

The structure of the sort and aggregate algorithms is similar, and we can merge them together into a single algorithm as shown in Algorithm 3.4. This solver can run on arbitrary covers with hierarchical schemas, sorting them first before computing the

Algorithm 3.4 Merged tree-sort and aggregate computation for path in Δ_H

```
1: ▷ Sort and compute aggregate for path A_p in Δ_H in range [from, to) over cover K
2: ▷ Initial call: SolveHierPath(K, root attributes of Δ_H, 0, K.size, aggAttr)
3: procedure SolveHierPath(K, A_p, from, to, aggAttr)
4:     ▷ values for ancestors of A_p in Δ_H are equal in range [from, to)
5:     ▷ sort on attributes A_p
6:     attrsToSort ← attributes in subtree of Δ_H rooted at first attribute of A_p
7:     sort block with rows [from, to) and columns attrsToSort on attributes A_p

8:     aggregate ← 0
9:     for each block [blockFrom, blockTo) of distinct values for A_p in [from, to) do
10:        if aggAttr in A_p then
11:            ▷ get value of aggregate attribute in current block
12:            ▷ use tuple at index blockFrom
13:            blockAggregate ← π_aggAttr K[blockFrom]
14:        else
15:            blockAggregate ← 1
16:        for each child path C_p of A_p in Δ_H do
17:            childAggregate ← SolveHierPath(K, C_p, blockFrom, blockTo, aggAttr)
18:            blockAggregate ← blockAggregate * childAggregate
19:        ▷ computed aggregate for block, add to total aggregate
20:        aggregate ← aggregate + blockAggregate
21:    return aggregate
```

aggregate. Merging the two steps together results in improved memory locality, as on each partition in the cover for the recursive steps of the algorithms both steps (sorting and computing the aggregate) are executed consecutively, whereas before both algorithms would operate over the entire cover relation after each other.

3.3.3 Parallelising the Solver

To parallelise the hierarchical solver, one must first sort the cover on the root attributes of its hierarchical variable order Δ_H. Sorting over the root attributes is a regular sort and can be done in parallel with existing multithreaded sorting algorithms. When the cover is sorted on its root attributes, split the cover relation into P blocks of disjoint root attributes, where P is the level of parallelism, and let each thread solve a single block on the children of the root attributes in Δ_H using the sequential algorithm of the hierarchical solver. This second step does not require any synchronisation between the threads, therefore, if there is a balanced division of work the achieved performance increase due to parallelism is near optimal for this step. Sorting on the root attributes in parallel is the main performance bottleneck, where

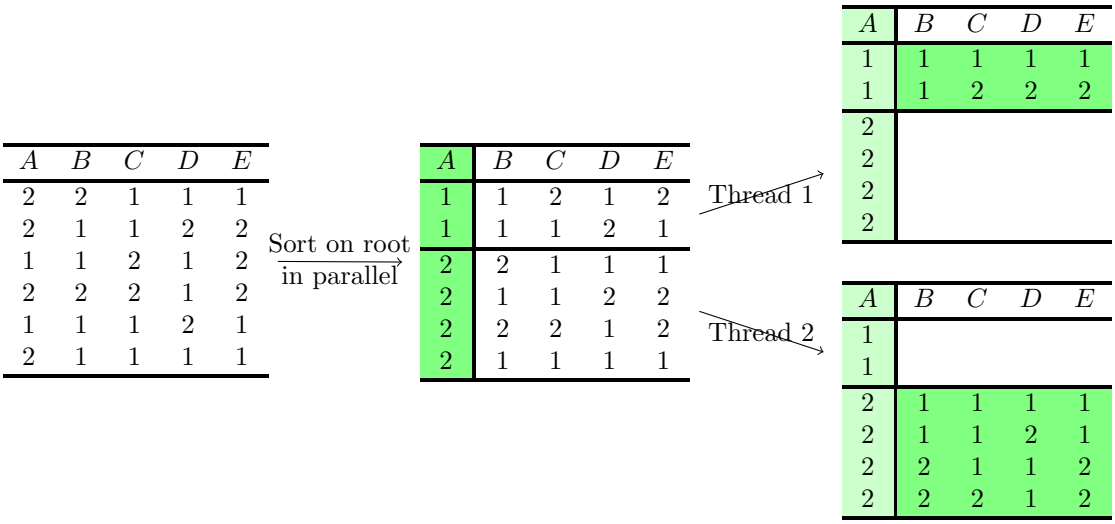


Figure 3.4: Division of work when parallelising the hierarchical solver.

the speed-up does not scale exactly linearly as the number of threads increases. Figure 3.4 shows how our example cover can be solved using two threads. After sorting the cover on root attribute A in parallel, split the relation into two blocks: one for $A = 1$ and one for $A = 2$, and let the two separate threads run the sequential algorithm on their respective block. Note that this example does not show a fair division of work, but larger covers often allow us to divide the work in a much more balanced way as we will see in the experiments. However, one can imagine a dataset with only a handful of different root-values, where separating them into P blocks will give skewed workloads for the different threads. It is encouraged to improve the algorithm such that it recursively splits the block for the same root values across multiple threads for skewed workloads, merging together the resulting aggregates afterwards.

One can easily extract a parallelised version of just the tree-sorting algorithm from the above by removing the steps to compute the aggregate.

Algorithm 3.5 Parallelised hierarchical solver

```
1: ▷ Compute aggregate for path A_p in Δ_H over cover K with parallelism P
2: procedure SolveHierParallel(K, Δ_H, P, aggAttr)
3:     root ← root attributes of Δ_H
4:     sort K in parallel (using P threads) on root
5:     aggregate ← 0
6:     for i in 1..P do
7:         do work in separate thread:
8:             ▷ find block of work for thread, balancing amount fairly between threads
9:             ▷ union of blocks for each thread covers the entire relation K
10:            [threadFrom, threadTo) ← disjoint block of root values in K
11:            localAggregate ← 0
12:            for each block [blockFrom, blockTo) of distinct values for root in
    [threadFrom, threadTo) do
13:                if aggAttr in root then
14:                    blockAggregate ← π_{aggAttr} K[blockFrom]
15:                else
16:                    blockAggregate ← 1
17:                for each child path C_p of root do
18:                    ▷ call sequential algorithm
19:                    childAgg ← SolveHierPath(K, C_p, blockFrom, blockTo, aggAttr)
20:                    blockAggregate ← blockAggregate * childAgg
21:                localAggregate ← localAggregate + blockAggregate
22:            ▷ when done, collect result in shared aggregate sum
23:            ▷ synchronised properly, of course
24:            aggregate ← aggregate + localAggregate
25:    wait for threads to finish
26:    ▷ each thread has added their local aggregate to aggregate
27:    return aggregate
```

Chapter 4

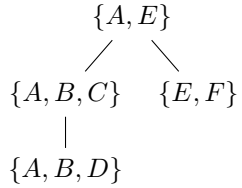
Aggregates over Covers with Arbitrary Acyclic Schemas

In Chapter 3 we have seen how to solve aggregates over covers with hierarchical schemas. Here we extend this approach to arbitrary acyclic schemas by decomposing an acyclic schema into a graph of hierarchical sub-schemas. We show that it is possible to construct such a hierarchical decomposition for any acyclic schema.

4.1 Decomposing into Hierarchical Sub-Schemas

Let C be an acyclic decomposition, and T be a tree for C satisfying the Coverage and Connectivity properties (also called a join-tree) in Definition 2.1. Let R be the root bag in T. To construct a hierarchical decomposition, we start by creating a hierarchical sub-schema H_{root} containing only R. For all children in T of bags in H_{root}, one can add them to H_{root} if the resulting schema remains hierarchical. There can be multiple possibilities to group bags into a hierarchical schema, as seen in Example 4.1.

Example 4.1. Consider decomposition $C = \{\{A, B, C\}, \{A, B, D\}, \{A, E\}, \{E, F\}\}$, with join-tree T:



There are four possible hierarchical sub-schemas that include root bag $\{A, E\}$:

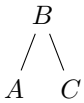
$\{\{A, E\}\}$ $\quad$ $\{\{A, E\}, \{A, B, C\}\}$

$\{\{A, E\}, \{A, B, C\}, \{A, B, D\}\}$ $\quad$ $\{\{A, E\}, \{E, F\}\}$

Once a hierarchical sub-schema H_{root} has been selected for the root bag, we create a new sub-schema for each child S of H_{root} in T. Similar to how we created the root sub-schema, one can add bags to S such that the sub-schema remains hierarchical, only with the additional constraint that the join attributes between the child S and its parent sub-schema in T must be in the root attributes of the hierarchical sub-schema of S. In a completed hierarchical decomposition, every bag belongs to exactly one hierarchical sub-schema.

Except for H_{root}, each sub-schema H in the decomposition must export its aggregate to its parent hierarchical sub-schema H_p. This export is a mapping from the join attributes between the two hierarchical sub-schemas to the aggregate for each join value in the sub-tree of hierarchical sub-schemas rooted at H. A sub-schema H can only export for attributes in its root attributes, since every bag in H must have the attributes to export in order to collect it through one scan over the cover using only $O(1)$ memory.

Example 4.2. Consider hierarchical schema $H = \{\{A, B\}, \{B, C\}\}$, with hierarchical variable order Δ_H as shown below, and root attribute B.



If one wishes to export the aggregate over non-root attribute C, it would not be possible to do this in one scan over the cover using $O(1)$ additional memory. Assuming that the cover is sorted on Δ_H, for each distinct tuple in the projection of $\{B, C\}$ over the cover one needs to know the number of distinct tuples in the projection of $\{A, B\}$ that have the same join value for B in order to compute the aggregate. However, this is only known after completing the scan over the block of tuples for that specific B. One solution for this specific query is to keep track of distinct values of C seen in the current block of B and process them all after having scanned all values of A for this B as well, however, this requires more than $O(1)$ memory and quickly gets more complex the deeper the attributes to export are in the hierarchical variable order.

Proposition 4.3. *Every acyclic schema C allows for a hierarchical decomposition to be constructed.*

Proof. The naive hierarchical decomposition is to consider every bag in C as its own hierarchical sub-schema, constructing the decomposition according to a join tree T of C. Trivially, every sub-schema has the join attributes with its parent sub-schema in its root attributes, making this decomposition always valid. □

Most decompositions can be decomposed into hierarchical sub-schemas in many different ways. One can consider different join trees for a decomposition, as well as using different nodes in these join trees as the starting root, or choosing different groupings of bags into hierarchical sub-schemas for a particular join tree, to construct a hierarchical decomposition. Therefore, it is encouraged to do more work on finding out how to select the best hierarchical decomposition.

4.1.1 Computing Aggregates over Hierarchical Decomposition

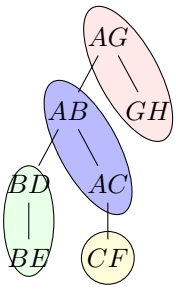
To solve aggregate queries over a hierarchical decomposition we show how to adapt the solver for hierarchical schemas to solve each of the sub-schemas in a hierarchical decomposition.

4.1.1.1 Tree-Sorting for Hierarchical Sub-Schema

Tree-sorting requires a variable order Δ over the decomposition of the cover such that for each bag in the decomposition its attributes lie along the same root-to-leaf path. A hierarchical variable order Δ_H of a sub-schema H is not sufficient in the sense that it misses attributes of the cover that are not in H. Therefore, we will show how to construct a variable order Δ over all attributes of the cover for an arbitrary hierarchical sub-schema, such that its hierarchical variable order Δ_H is at the root of Δ. Sorting over Δ is will then cause the cover to be sorted over Δ_H as well.

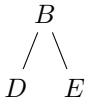
When considering the join tree T of an acyclic decomposition C, the bags of the hierarchical sub-schema H are joined with one parent relation if it is not the root sub-schema, as well as joining with possibly a number of child sub-schemas. For each child, one can construct a variable order containing the join attributes with H at the top of its root attributes. Place this variable order, minus the join attributes, under the root-to-leaf path in Δ_H of the bag in H this child joins with. All attributes except the join attributes in the variable order of the child are not present outside of the sub-tree of bags rooted at the child, by Connectivity of the join tree. Similarly, when H joins with a parent bag in T, construct a variable order for the "upwards" sub-tree in T rooted at the parent bag it joins with, remove the join attributes with H at the top of the root attributes of this variable order, and place it under the join attribute lowest in Δ_H. Example 4.4 illustrates this process.

Example 4.4. Consider decomposition $C = \{\{A, B\}, \{A, C\}, \{B, D\}, \{B, E\}, \{C, F\}, \{A, G\}, \{G, H\}\}$ with the following join tree T:



The coloured markings represent the hierarchical decomposition, where each marking is a hierarchical sub-schema. Let us focus on sub-schema $H = \{\{A, B\}, \{A, C\}\}$ (marked in blue). Constructing a variable order Δ over all attributes in C such that Δ_H of H is at the root of Δ, we first create variable orders for the subtree of bags in T for each child joining with H.

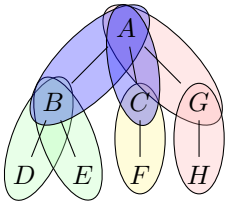
For the sub-tree of child BD, we receive the following variable order, with join attribute B as its root attribute:



For child bag CF we obtain:

Finally, sub-schema H also joins with the parent bag AG in T. The "upwards" sub-tree of AG in T admits the following variable order with join attribute A as its root attribute:

Position the variable orders for the children under the path of the bag in H they are joined with: variable order of child BD is placed under path $A-B$ in Δ_H, variable order for CF under $A-C$. Place the variable order of the parent of H under the join attribute lowest in Δ_H (which is A). The result is the following variable order with Δ_H at its root:



The coloured markings represent bags in the hierarchical sub-schemas of that colour.

While the tree-sorting algorithm can be applied over the constructed variable order Δ to get a cover sorted on Δ_H, it performs more work than necessary. Whenever the tree-sorting algorithm reaches attributes in Δ that are not in Δ_H, it can stop and ignore these attributes and their descendants. The resulting cover will still represent the same relation, but will not be sorted on attributes other than the ones in Δ_H. The attributes not in Δ_H are only relevant for detecting which attribute values to move when sorting on one of the attributes in Δ_H such that the cover structure remains intact.

For the variable order Δ of the previous example this results in the tree-sorting algorithm only sorting three attributes for sub-schema H: A, B, and C. When sorting on A, all attributes are moved. Sorting on B only moves attributes B, D, and E, while sorting on C moves C and F.

4.1.1.2 Computing the Aggregate of a Hierarchical Sub-Schema

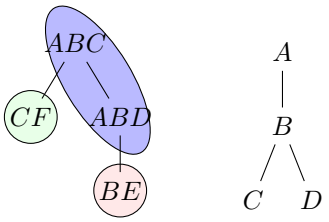
Computing an aggregate over a hierarchical sub-schema is then similar to computing an aggregate over a full hierarchical schema, with the following changes:

Export to parent A sub-schema H exports its aggregate in a mapping over the join attributes with its parent sub-schema. These join attributes must be in the root attributes of Δ_H of H, therefore we get a slightly different algorithm for processing the aggregate for the root path in Δ_H. Instead of summing the aggregates for each block of distinct values for the root attributes, we compute the export key by taking the values for the join attributes in the block (remember these are equal for each tuple in the block) and add the aggregate of a block under its export key in the export mapping. The result will then be the export map containing aggregates mapped over join values, instead of a single aggregate sum. Processing of non-root paths can keep collecting a single aggregate sum.

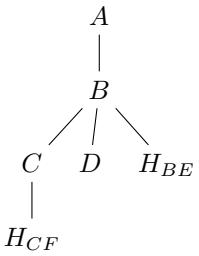
One exception is the root sub-schema of the hierarchical decomposition. Since this sub-schema does not export to any parent sub-schema it can be solved using the algorithm for non-root paths as well. This benefits the performance, as simple summations are more efficient than aggregating to a map.

Children in hierarchical decomposition Sub-schemas might have a number of child sub-schemas (dependencies) in the hierarchical decomposition. While computing the aggregate for a sub-schema H one should join its aggregate with the exported aggregates of its dependencies. For each dependency, join its aggregate in the top-most path in Δ_H of H such that all join attributes with the dependency are contained in the attributes of that path plus its ancestors.

Example 4.5. Consider (acyclic) decomposition $C = \{\{A, B, C\}, \{A, B, D\}, \{B, E\}, \{C, F\}\}$ with the following hierarchical decomposition (left) and hierarchical variable order Δ_H for sub-schema $H = \{\{A, B, C\}, \{A, B, D\}\}$ (right):



The aggregate of child sub-schema $H_{CF} = \{\{C, F\}\}$ of H is joined at the path containing attribute C in Δ_H. The aggregate of dependency $H_{BE} = \{\{B, E\}\}$ is joined at path containing attributes $\{A, B\}$. This will be denoted as follows:



Sum aggregate attribute If the aggregate query is a sum over an attribute A, one should be careful not to aggregate the values for A multiple times, as A might be part of multiple sub-schemas. Only one sub-schema should take the values for A into account for the aggregate. A simple solution is to let the top-most sub-schema process the values for the aggregate attribute.

The algorithms for solving a root path and non-root path of a sub-schema are given in Algorithm 4.6 and Algorithm 4.7 respectively, and are adaptations of Algorithm 3.4 that implement the changes described above.

4.1.2 Optimisations

In contrast to computing the aggregate over a fully hierarchical schema, sub-schemas require a number of additional computational steps to solve aggregates. These include

Algorithm 4.6 Merged tree-sort and aggregate computation for root path in Δ_H

▷ Sort and compute aggregate map on export schema S_{exp} for root path A_p in Δ_H over cover K
▷ Let Δ be the variable order over the full decomposition with Δ_H at its root
▷ Let S_{exp} be the export schema for this sub-schema to its parent sub-schema.
▷ Initial call: SolveHierSubSchemaRoot(K, root attrs of $\Delta_H, S_{exp}, aggAttr$)
procedure SolveHierSubSchemaRoot($K, A_p, S_{exp}, aggAttr$)
    ▷ sort on attributes A_p
    $attrsToSort \leftarrow$ attributes in subtree of Δ rooted at first attribute of A_p
    sort entire cover on attributes A_p, only moving values for attributes $attrsToSort$

    $export \leftarrow \{\}$ ▷ Export for this sub-schema
    for each block $[blockFrom, blockTo)$ of distinct values for A_p in the cover **do**
        if $aggAttr$ in A_p and sub-schema is top-most containing $aggAttr$ **then**
            ▷ get value of aggregate attribute in current block
            ▷ use tuple at index $blockFrom$
            $blockAggregate \leftarrow \pi_{aggAttr} K[blockFrom]$
        else
            $blockAggregate \leftarrow 1$
        for each child path C_p of A_p in Δ_H **do**
            $childAggregate \leftarrow$ SolveHierSubSchema($K, C_p, blockFrom, blockTo, aggAttr$)
            $blockAggregate \leftarrow blockAggregate * childAggregate$
        for each child sub-schema export $export_c$ joining with this path A_p **do**
            $childExportKey \leftarrow \pi_{S(export_c)} K[blockFrom]$
            $childExportAggregate \leftarrow export_c[childExportKey]$
            $blockAggregate \leftarrow blockAggregate * childExportAggregate$
        ▷ computed aggregate for block, add to export for export key of block
        $exportKey \leftarrow \pi_{S_{exp}} K[blockFrom]$
        $export[exportKey] \leftarrow export[exportKey] + blockAggregate$
    return $export$

aggregating to an export map, joining with aggregates of child sub-schemas, and an additional check whether the sub-schema should aggregate the attribute to sum.

In Algorithms 4.6 and 4.7 some of these steps have been implemented naively: checking whether to aggregate the attribute to sum is done in every recursive call, and it is not show in the algorithms how to get the child dependencies the current path should join with. Due to the recursive structure of the algorithms, doing these for every recursive call would be expensive. For each path in the hierarchical variable order Δ_H of the sub-schema one should compute its dependencies to join with and whether it must aggregate the sum attribute only once at the beginning, and make this information available for each recursive call.

Additionally, certain paths might have no dependencies to join with, or no child

Algorithm 4.7 Merged tree-sort and aggregate computation for non-root path in Δ_H

```
1: ▷ Sort and compute aggregate for path A_p in Δ_H in range [from, to) over cover K
2: ▷ Let Δ be the variable order over the full decomposition with Δ_H at its root
3: ▷ Initial call: SolveHierSubSchema(K, root attrs of Δ_H, 0, K.size, aggAttr)
4: procedure SolveHierSubSchema(K, A_p, from, to, aggAttr)
5:     ▷ values for ancestors of A_p in Δ_H are equal in range [from, to)
6:     ▷ sort on attributes A_p
7:     attrsToSort ← attributes in subtree of Δ rooted at first attribute of A_p
8:     sort block with rows [from, to) and columns attrsToSort on attributes A_p

9:     aggregate ← 0
10:    for each block [blockFrom, blockTo) of distinct values for A_p in [from, to) do
11:        if aggAttr in A_p and sub-schema is top-most containing aggAttr then
12:            ▷ get value of aggregate attribute in current block
13:            ▷ use tuple at index blockFrom
14:            blockAggregate ← π_aggAttr K[blockFrom]
15:        else
16:            blockAggregate ← 1
17:        for each child path C_p of A_p in Δ_H do
18:            childAggregate ← SolveHierPath(K, C_p, blockFrom, blockTo, aggAttr)
19:            blockAggregate ← blockAggregate * childAggregate
20:        for each child sub-schema export export_c joining with this path A_p do
21:            childExportKey ← π_S(export_c) K[blockFrom]
22:            childExportAggregate ← export_c[childExportKey]
23:            blockAggregate ← blockAggregate * childExportAggregate
24:        ▷ computed aggregate for block, add to total aggregate
25:        aggregate ← aggregate + blockAggregate
26:    return aggregate
```

paths in the hierarchical variable order. In these cases, one can completely remove their respective empty for-loops, which would otherwise have been executed up to $O(n)$ times performing loop checks over empty sets, where n is the number of tuples in the cover. The final implementation used in the benchmarks takes advantage of inlining function calls[1] to generate eight different flavours of solving a path in a sub-schema, covering all possible combinations of the following three binary options:

- Whether to include the value for the sum attribute in the current path;
- Whether the path should join with aggregates of dependencies;
- Whether the path has any child paths in Δ_H.

The recursive structure of the algorithm results in many function calls as well, causing lower instruction locality. Therefore it is strongly encouraged to apply techniques such as code compilation to generate a single block of code to solve a specific hierarchical sub-schema, as we expect that this will have a notable effect on performance.

4.1.3 Parallelisation

Parallelising Algorithms 4.6 and 4.7 is done in exactly the same manner as for the solver for hierarchical queries in Algorithm 3.5. First the cover is sorted on its root attributes in parallel, after which the cover relation is split into P blocks of disjoint root attributes, where P is the level of parallelism. Each of the P threads then solves its assigned block sequentially. Merging the results is slightly different when solving sub-schemas that export an aggregate map instead of a single aggregate sum: we now need to combine all key-value pairs in the export of each thread into one single mapping. This is done sequentially by a single thread in our implementation, as it takes only a tiny fraction of the total time for the datasets we benchmark on. One could optionally parallelise this step as well if it is detected that doing this sequentially has a performance impact on certain datasets.

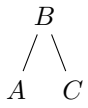
4.1.4 Repeated Tree-Sorting

Applying the tree-sorting algorithm over a cover that is already sorted on a variable order Δ can preserve some of the sorted structure of Δ and skip sorting over certain

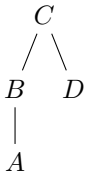
[1] `https://kotlinlang.org/docs/reference/inline-functions.html`

attributes of the new variable order to sort on, if the regular sorting algorithm used in tree-sorting is stable.

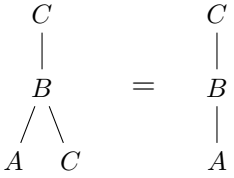
Example 4.6. Consider a cover with acyclic decomposition $C = \{\{A, B\}, \{B, C\}, \{C, D\}\}$ that is sorted on:



Sorting this cover over the variable order

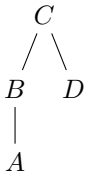


allows for attributes B and A to be skipped in the tree-sorting algorithm, since these will already be sorted by preservation of the structure of the initial tree the cover was sorted on. Starting by sorting on attribute C, moving all attributes A, B, C, and D, results in the cover being sorted on:



Note that we require a stable sorting algorithm, as tuples with the same value for C must remain in the same relative ordering as they originally were in.

We see that sorting on C keeps attributes B and A sorted as well, in the exact order we would like them to be. There is no need to sort on these attributes, so what remains is sorting over D such that we get:



This optimisation for repeated tree-sorting can be beneficial for solving the separate sub-schemas in a hierarchical decomposition, each requiring you to sort the cover over their hierarchical variable order. Since these hierarchical variable orders can contain overlapping attributes, we might be able to skip sorting over certain attributes that are already sorted. Additionally, we saw that covers constructed by the proposed implementation of the cover-join algorithm are sorted on some variable

order. Therefore, we can apply this optimisation to the first hierarchical sub-schema, or even a fully hierarchical schema as well.

This repeated sorting optimisation is not implemented in the benchmarked algorithms in section 6, but it is highly encouraged to implement this and see how it influences the performance.

Chapter 5

Implementation

This chapter describes the implementation of the different solvers benchmarked in the experiments, including naive solvers over the flat join result, solvers using eager aggregation, and the hierarchical solver.

The solvers have been implemented to support the count and sum aggregate queries:

- `SELECT SUM(1) FROM R;` where `R` is a natural join in the form of $R = R_1 \bowtie \cdots \bowtie R_n$.
- `SELECT SUM(attr) FROM R;` where `attr` is an attribute in $S(R)$.

The datasets these solvers are executed on consist of multiple tables joined together through an acyclic natural join, and have a cover that represents the full join result. The decomposition of the cover consists of the schemas of the relations in the join.

While implementing these solvers, special attention has been given to prevent the boxing and unboxing of primitive values on the JVM in hot sections of our code[1].

5.1 Naive Solvers over Full Join Result

Std, StdSrt The naive approach to solving the aggregate queries is to operate over the full join result R of the separate tables, accumulating the aggregate through a basic scan over R. For the count and sum aggregate queries sorting is not required, however, more advanced aggregate queries often perform sorting as one of their intermediate steps. For instance, to solve group-by aggregate queries the join result is first sorted on the group-by attributes such that all tuples belonging to the same

[1] `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`

group are in one consecutive block, ready to be processed further. Because of the use of sorting it is beneficial to see the impact of sorting on the full join result. Therefore we will benchmark two flavours of the standard solver. One that does not apply a sorting step (**Std**), capable of solving the basic queries, and one that first sorts the full join result on the aggregate attribute of the query (**StdSrt**). Both solvers receive the materialised join result as input. See Algorithm 5.8 for the implementation of these two solvers.

Algorithm 5.8 Solver STD and STDSRT

```
1:  procedure AGGREGATE(R, doSort)        ▷ Compute aggregate over join result R
2:      if doSort then
3:          Sort R on attr                          ▷ Sort over the aggregate attribute
4:      aggr ← 0
5:      for each row in R do
6:          if aggregate query is sum over attr then
7:              rowAggr ← π_attr(row)                                  ▷ SUM(attr)
8:          else
9:              rowAggr ← 1                                            ▷ SUM(1)
10:         aggr ← aggr + rowAggr
11:     return aggr
```

5.2 Eager Aggregation Solvers

EagerC, EagerT An existing approach to efficiently solve aggregate queries over a natural join is to push the aggregate past the join, called eager aggregation [13]. Not having to materialise the natural join, it is able to run on datasets where the join result blows up, but where the separate tables are relatively small. It gives us an approach to compare against with the hierarchical solver over covers. The solvers receive a join-tree as input, and for each bag in this tree it will construct an aggregate export map for its parent bag while joining with the export maps of its children. Starting from the leafs of the tree these export maps are constructed upwards until finally the root of the tree has been processed, exporting the final aggregate result over the complete join.

Two approaches for eager aggregation have been implemented, each with their own benefits. The first approach (**EagerC**) operates over the cover by taking the projection over the cover for each bag in the join. A disadvantage of taking these projections over the cover is that the actual size of the projection for this bag might

be much smaller than the number of tuples in the cover. Therefore, scanning the entire cover might be less efficient than scanning the table for the bag directly, as a lot of duplicate entries will be encountered in this case. On the other hand, it does profit from the properties guaranteed by a cover, such as being free of dangling tuples [6]. If the separate tables contain many of these non-contributing tuples, processing over the cover might be more favourable as it has filtered out these tuples already. The solver is implemented according to Algorithm 5.9.

The other approach for eager aggregation (**EagerT**) is similar but executes over the separate tables in the join instead of over a cover. Depending on the size and quality of these tables it can be very efficient if the join contains a few large and many small tables. The performance can, however, be impacted by the existence of dangling tuples. We will see various types of datasets that illustrate these effects. This solver is implemented similarly to EagerC in Algorithm 5.9, except that for each bag it scans its corresponding table directly instead of taking the projection over the cover.

5.3 Hierarchical Solver

Hier This solver is the implementation of the proposed solver in the previous chapters, supporting both hierarchical (Algorithm 3.4) and non-hierarchical queries (Algorithms 4.6 and 4.7), as well as parallelisation (Algorithm 3.5 and Section 4.1.3). We call a query hierarchical if its cover has a hierarchical schema. The solver receives a hierarchical decomposition as input, which will be of size 1 for hierarchical queries. You can find the exact decompositions used for each dataset in Appendix A.

Algorithm 5.9 Solver EagerC

```
1: ▷ Compute aggregate over cover relation K with join-tree T
2: procedure Aggregate(K, T)
3:     root ← root of T                                  ▷ Root bag in the join-tree
4:     rootExport ← SolveNode(root, K)
5:     ▷ rootExport has only one key-value pair with the empty schema {} as key
6:     ▷ Total aggregate has been accumulated under this key
7:     return rootExport[{}]

8: ▷ Compute aggregate export for node in join-tree over cover relation K
9: procedure SolveNode(node, K)
10:    ▷ Recursively compute export maps for children of node
11:    children ← children of node in join tree
12:    childExports ← children.map(child → SolveNode(child))
13:    nodeExport ← empty map {}          ▷ Will store result export map for node
14:    S ← schema of bag represented by node

15:    ▷ Retrieve the export schema S_E of node
16:    if node has parent P then
17:        S_P ← schema of P
18:        S_E ← S_P ∩ S
19:    else
20:        S_E ← {}                                      ▷ No parent, empty schema

21:    for each tuple in K do
22:        bagTuple ← π_S(tuple)                         ▷ Projection of S over tuple
23:        if bagTuple already seen then
24:            continue
25:        if attr ∈ S and attr ∉ S_P then
26:            ▷ node is top-most node in join-tree that contains aggregate attribute
27:            aggregate ← π_attr(bagTuple)
28:        else
29:            aggregate ← 1
30:        for each childExport in childExports do
31:            S_childExport ← S ∩ S_child          ▷ S_child is schema of child childExport
32:            childExportKey ← π_{S_childExport}(bagTuple)
33:            childAggregate ← childExport[childExportKey]
34:            aggregate ← aggregate * childAggregate
35:        ▷ Add the aggregate to any existing aggregate for exportKey
36:        exportKey ← π_{S_E}(bagTuple)
37:        nodeExport[exportKey] ← nodeExport[exportKey] + aggregate
38:    return nodeExport
```

Chapter 6

Experimental Evaluation

This chapter includes several experiments that compare the performance of the proposed hierarchical solver against a number of different aggregate query solvers, including the naive approach to solve over the full join result, as well as two flavours of aggregate solvers that use eager aggregation [13] and operate over the separate tables of the natural join query or the cover directly. The experiments show that the proposed hierarchical solver outperforms its best competing solver by up to 37% on hierarchical queries. It can outperform the standard solvers by orders of magnitudes, where the performance gap is similar to the difference in size between the cover and the fully materialised join. For non-hierarchical queries the hierarchical solver is no longer the best performing solver, slowly degrading as more subqueries require to be solved. Additionally, we show that using the cover-join algorithm to construct the cover has a large impact on the hierarchical solver, as it produces a cover nearly sorted for hierarchical queries. Finally, we benchmark the parallelised hierarchical solver and see that speedup is near linear for hierarchical queries. A variety of datasets is used, each with their own unique properties, showing where the hierarchical solver performs at its best and where it is outperformed by competing solvers.

Please refer to Appendix A for details about the datasets used in the experiments, such as sizes and table schemas. For the experiment set-up such as machine type and software versions see Appendix B.

6.1 Hierarchical Queries - Housing

The Housing dataset is the only dataset we experiment on that consists of a natural join that is hierarchical. The join of its six tables is a star-join over the `postcode` attribute.

Dataset	Full join size (tuples)	Cover size (tuples)	Compression factor
Housing-12	64,756,800	300,000	x216
Housing-20	399,500,000	500,000	x799
Housing-100	75,000,000,000	2,500,000	x30,000
Housing-200	700,000,000,000	5,000,000	x140,000
Housing-500	12,500,000,000,000	12,500,000	x1,000,000

Table 6.1: Sizes of different Housing dataset scales.

There are multiple scaled sizes of the Housing dataset, Table 6.1 showing the sizes of the join as well as the cover for the different scale levels that are benchmarked here. The compactness of a cover compared to the size of a full join is clearly visible for the Housing dataset. None of the benchmarks for the Housing datasets were influenced by garbage collection events, none occurred during the timed runs.

The difference in performance between the solvers on a count query versus a sum query is minimal, as seen in Figure 6.2. An exception is the Std solver, which shows to perform up to 6 times as fast on a count query. The reason being that the count query is solved using a basic for-loop, summing over the n tuples in the relation, adding 1 to the aggregate sum for each tuple. The solver does not access any data of the relation, and is therefore very fast. Other solvers still need to access the data for count queries, mainly through sorting or hashing, and therefore do not see such a notable increase in performance. Since most types of aggregate queries will operate over the data itself, the count query being an exception, as well as the non-Std solvers having similar performance on either query type, we will execute all our following experiments on sum aggregate queries.

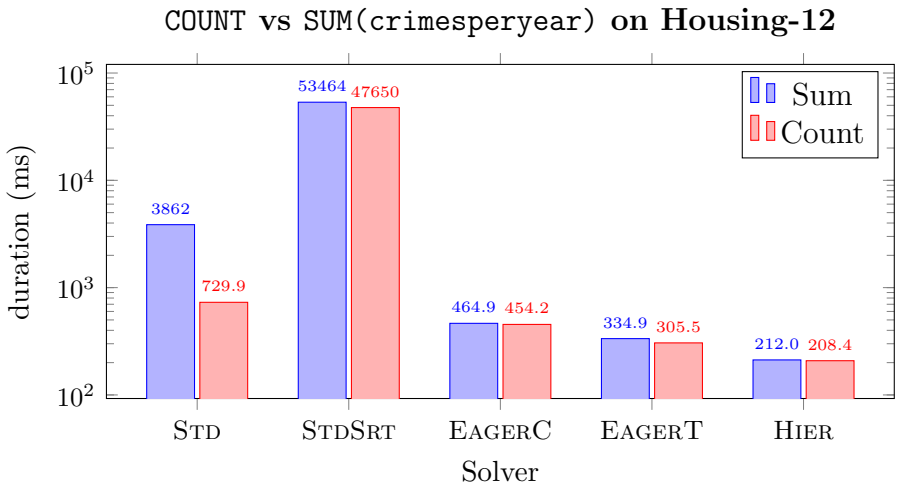


Figure 6.2: Comparison performance solvers on Count vs Sum query.

The attribute to sum over does not influence the performance of any solver, this is therefore chosen arbitrarily. For the Housing datasets we sum over the `crimesperyear` attribute.

Housing-12 The materialised join is too large for most of the Housing datasets benchmarked in this section, therefore STD and STDSRT are excluded from the results of these larger datasets. The smallest Housing dataset in our experiments, Housing-12, does still allow us to compute the full join and run these solvers on. The results for this dataset are found in Figure 6.2. Note the log scale for the y-axis. Due to the large size of the fully materialised join the standard solvers are significantly slower than the eager and hierarchical solvers, with STDSRT being around 250 times slower than HIER. This gap in performance nicely follows the gap between the cover size and full join size, where the cover has a compression factor of 216.

Looking at the three non-standard solvers, HIER clearly outperforms the other two, being more than twice as fast as EAGERC and around 37% faster than EAGERT. The Housing datasets have no dangling tuples, apart from a tiny amount in the Institution table (around 1% of the tuples in the table). This is the best setting for EAGERT, and even then it is outperformed by HIER quite significantly. EAGERC is the slowest of the three, explained by four tables in the dataset having a much smaller size than the size of the cover relation, causing EAGERC to scan over a lot of duplicate tuples when taking the projections of these tables over the cover.

Housing-20 to Housing-500 The non-standard solvers (EAGERC, EAGERT, and HIER) are executed on the larger datasets 20, 100, 200, and 500 as well. Figure 6.3 shows the performance results for these solvers. We see consistent results similar to the ones in Housing-12, showing that the performance improvement for HIER over the eager solvers stays the same as the dataset grows in size.

What is interesting is that even though EAGERT and HIER each process 232,075,000 (sum of tables) and 337,500,000 (cover) data values respectively in Housing-500, EAGERT having to process around 30% less data, HIER still outperforms the former. Benefiting from processing with high memory locality, HIER is able to do all of the final aggregate computation in a single scan over the cover, only accumulating the aggregate through simple summations, whereas EAGERT processes separate tables, constructing hash maps to aggregate each table.

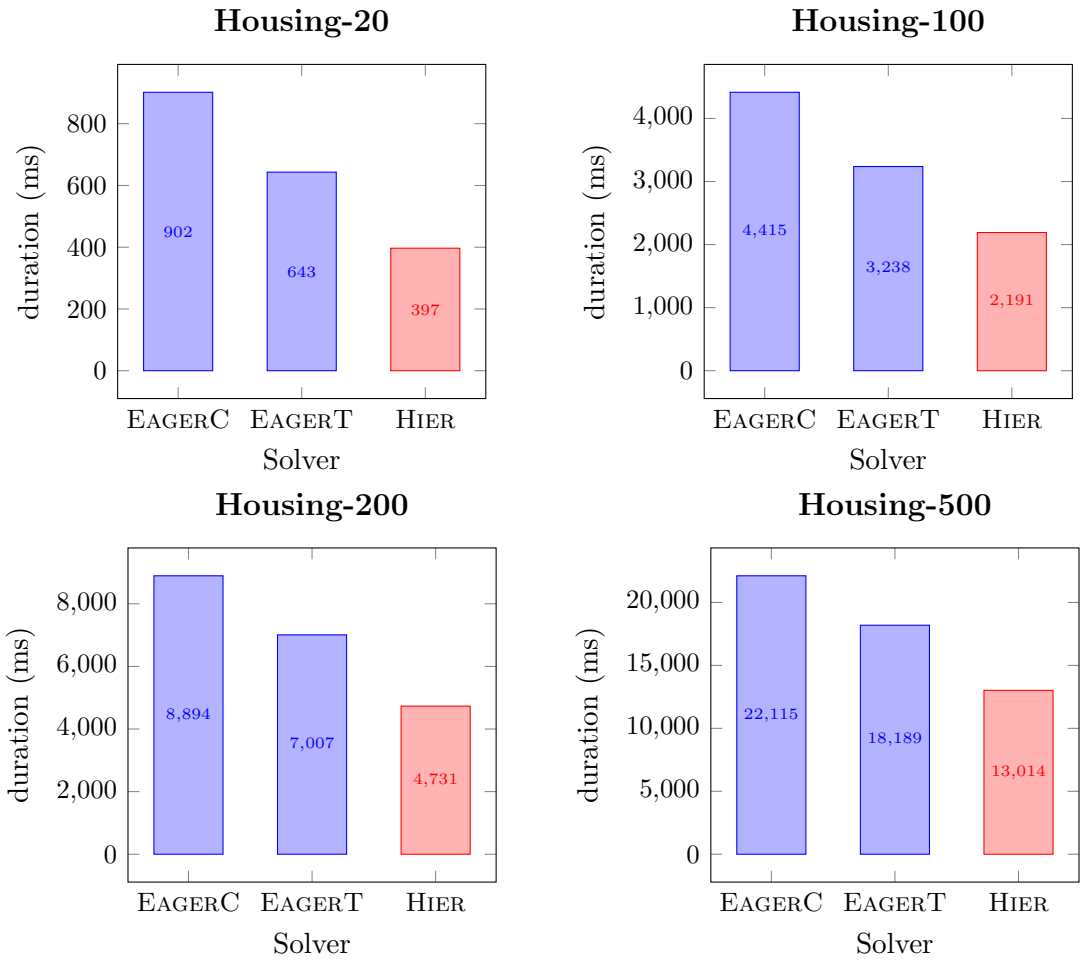


Figure 6.3: Performance results on the Housing datasets.
`SELECT SUM(crimesperyear) FROM R;`

6.2 Non-Hierarchical Acyclic Queries

We have seen that our approach performs well for aggregate queries over hierarchical joins. We also showed how to extend our solver to support non-hierarchical queries by constructing a hierarchical decomposition in the previous chapter. In this section we will benchmark the solvers on a number of non-hierarchical queries to show the cost for Hier having to process multiple hierarchical subqueries. Additionally, we will run the solvers on two types of datasets of which the covers are the same size as the full join result to see how much overhead the hierarchical solver has over the naive approach.

6.2.1 LastFM

The LastFM dataset has a relatively small cover with 132,070 tuples that represents a much larger join result of 59,079,380 tuples (a compression factor of 447). Unique about this dataset is the high amount of dangling tuples in the separate tables. The two `Usertaggedartiststimestamps` tables consist of roughly 60% of these non-contributing tuples and the two `Userartists` tables even as much as 78%. One can see the impact of this in the performance of EagerT in Figure 6.4, which is around twice as slow as EagerC and Hier. The two standard solvers, especially StdSrt, are between one and three orders of magnitude slower than the three non-standard solvers. This difference in performance again is similar to the gap between the cover size and join size. Hier now is slightly slower than EagerC, contrary to the results on the Housing datasets. Since LastFM is not hierarchical, we need to solve two hierarchical subqueries, requiring two separate sorts of the cover relation by Hier. We can conclude that even having only two subqueries can already be disadvantageous for the hierarchical solver, although it will still be the better option if the tables contain many dangling tuples over EagerT.

6.2.2 Retailer

The Retailer dataset is unique in that the cover is of the exact same size as the full join result. Therefore, we do not expect an improvement in performance, in fact it would be expected that Hier will be slower than the standard solvers due to overhead in processing over the cover. Running the experiment on this dataset is therefore useful to show us how much this overhead is in practice.

The smallest hierarchical decomposition for this dataset consists of three hierarchical subqueries. The selected decomposition for the experiments has the root

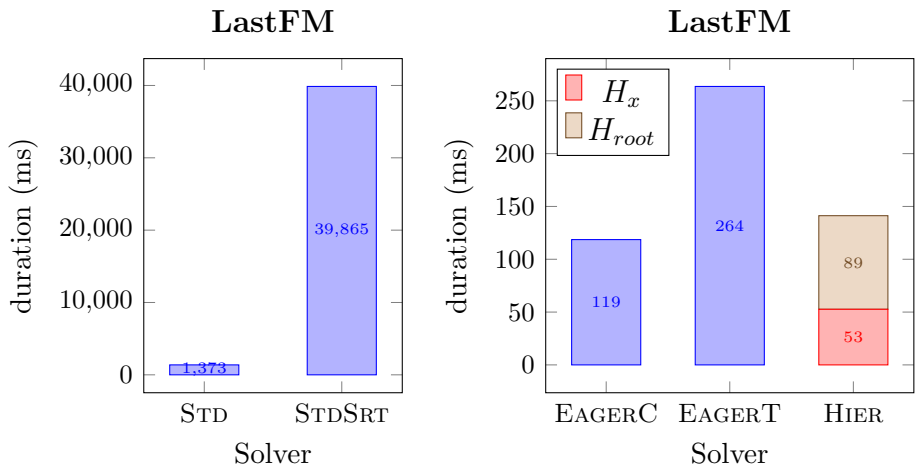


Figure 6.4: Performance results on the LastFM dataset.
Note the differently scaled y-axes.
`SELECT SUM(userid) FROM R;`

subquery made as large as possible, as an alternative would be to only solve this root hierarchical query using the Hier solver and create the exports for its child subqueries using eager aggregation to prevent having to sort the whole cover multiple times. Figure 6.5 shows the performance results of all solvers over the Retailer dataset. The durations to solve each of the subqueries in the decomposition for Hier have been marked separately. The two standard solvers are clearly the fastest, as expected, with EagerT coming close to the performance of StdSrt. The EagerC and Hier are the slowest of them all.

The tables in the Retailer dataset consists of one large table `promotedF` with 29,200,000 tuples, which is joined against 12 tiny tables each between 200 and 400 tuples. Both the cover as well as the full join have the same number of tuples as `promotedF`: 29,200,000. This large amount of small tables is far from ideal for both EagerC and Hier, as the cover fills up the columns for the attributes of these small tables with duplicate information. This additional data does not contribute to the result, but does slow down these solvers as we see in the results here.

Comparing Hier against EagerC we do see that the former is not far off in terms of performance, even though the cover is being sorted three times by the hierarchical solver for each of the subqueries in the hierarchical decomposition. H_{root}, the root hierarchical query, takes up most of the running time for Hier. This subquery also contains most of the work, it being the largest subquery as well as having to join with the results of its two children H_{day} and H_{sku}. All experiments on Retailer suffer slightly from the garbage collector due to the large relation sizes, where up to 10% of

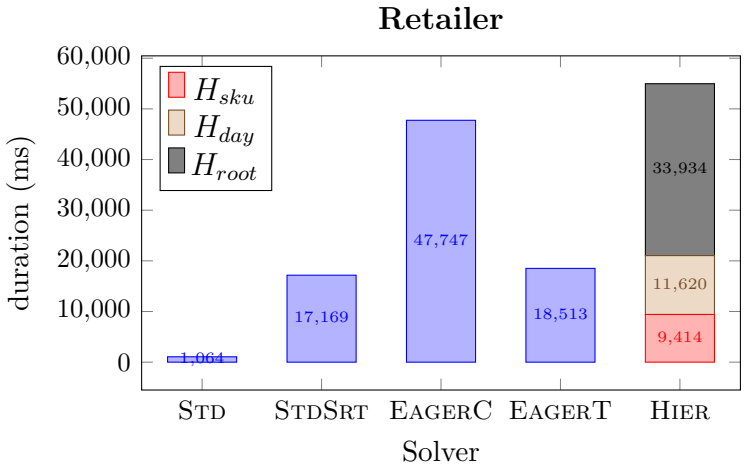


Figure 6.5: Performance results on the Retailer dataset.
`SELECT SUM(region) FROM R;`

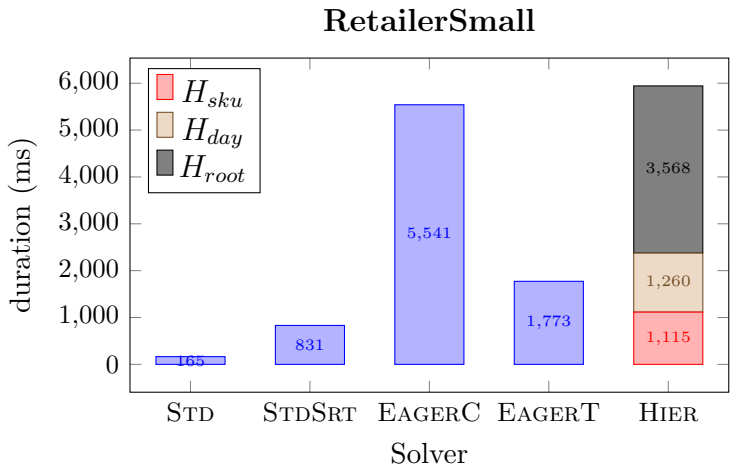


Figure 6.6: Performance results on the RetailerSmall dataset.
`SELECT SUM(region) FROM R;`

the time was spent being paused by the garbage collector.

We also benchmark the solvers on a smaller version of the Retailer dataset named, subsequently, RetailerSmall. The results of this can be found in Figure 6.6. Here we can see that StdSrt and EagerT have gained in relative performance over EagerC and Hier, but overall the results are comparable to the results of the larger Retailer dataset.

6.2.3 USRetailer

Finally, we have the USRetailer dataset, which is quite similar to the Retailer dataset in terms of properties. Its one large `Inventory` table is joined against four small

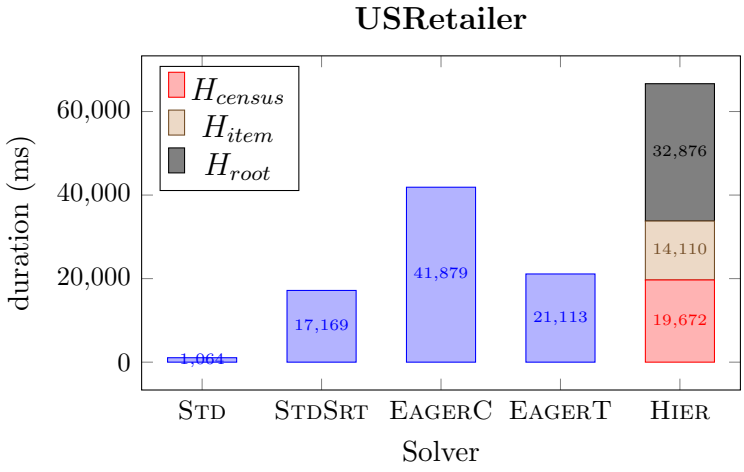


Figure 6.7: Performance results on the USRetailer dataset.

`SELECT SUM(prize) FROM R;`

tables, resulting in a join that is again of equal size as the cover. The smaller tables now contain a number of dangling tuples, although relative to the size of the large `Inventory` table this is an insignificant amount. Figure 6.7 shows the experiment results. Hier performs the worst here, being around three times as slow as EagerT and StdSrt. Even EagerC is around 40% faster, whereas in Retailer these had comparable performances. Again, every experiment on USRetailer is spending between 5% and 10% of its time on garbage collection.

The different datasets showed us that our hierarchical solver performs great on datasets with hierarchical join queries such that there is only a single hierarchical subquery to solve. For decompositions with more than one hierarchical subquery it starts to lose over the alternative query solving methods, although it does not degrade terribly. By looking at optimising repeated tree-sorting one could possibly reduce the time for solving multiple hierarchical subqueries in a decomposition, hopefully bringing it more in line with the performance of the eager aggregation solvers.

6.3 Shuffled Covers

The covers used in the experiments were produced by the cover-join algorithm, which constructs covers sorted on a particular sorting tree. For hierarchical joins these covers are nearly sorted according to the sort tree required to solve the hierarchical query. This greatly speeds up the sorting step, as little to no data needs to be reordered. To show how much shuffling a cover affects the performance we also

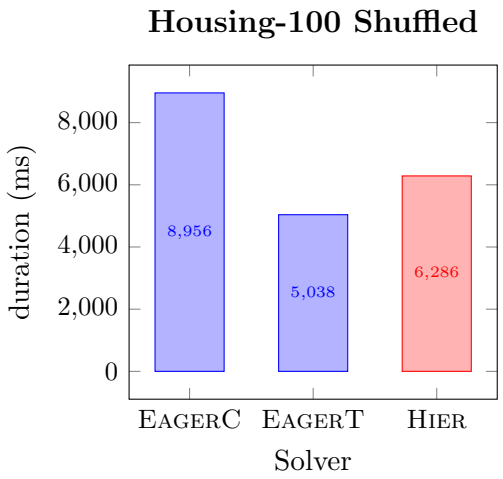


Figure 6.8: Performance on shuffled Housing-100 cover and tables.
`SELECT SUM(crimesperyear) FROM R;`

execute an experiment on the Housing-100 dataset with shuffled covers and tables. Shuffling a cover is more complex than shuffling a regular relation. Appendix C goes into more detail about this and shows how to shuffle a cover. Figure 6.8 indicates that Hier is impacted significantly, now being slightly slower than EagerT where before Hier used to be up to 30% faster. Noticeable is that all solvers are affected by the shuffling of the input data, not only the hierarchical solver. This is likely because of the hash maps filling up faster for each bag while scanning the table or projection, as all tuples, and therefore the export keys to be put in the map, are distributed equally now instead of appearing in sorted order throughout the scan. It is therefore more likely to discover a new export key in the early phase of the scan, filling up the aggregate export map with keys more quickly. Additionally, it becomes more difficult for branch prediction to consistently guess whether each tuple is unique or not.

The results here show that the chosen cover-join algorithm makes a big impact on the hierarchical solver in particular, and using a cover-join that produces an (almost) sorted cover is highly preferable.

6.4 Parallelism of the Hierarchical Solver

In the previous chapters we described how to parallelise the implementation for our hierarchical solver. In this section the hierarchical solver is benchmarked on the selected datasets, showing the speedup for increasing levels of parallelism. Not scaling exactly linearly, it still runs up to 5 times as fast in an 8-threaded environment when compared to running sequentially, and between 3-4 times as fast using 4 threads.

The experiments run on a machine which has 8 hyper-threaded CPUs. Using more than 8 threads does not benefit the performance, as the solvers are CPU-bound. Our experiments compare the total solve duration for $P = 2$, 4, and 8 threads against running the solver sequentially. The solvers are run on the same sum aggregate queries as in the previous sequential experiments.

The main bottleneck for the parallel implementation is the initial sort on the root attribute of the sorting tree for a hierarchical subquery. This sort needs to be completed first before we can split our relation into separate blocks for each thread. The time it takes to do this sort and the speedup compared to the sequential setting is indicated separately in the benchmark results, as they scale differently for higher P. After the cover is sorted on the root attributes, we can split the cover into separate consecutive blocks, one for each thread. There is no interference between threads, so our speedup should be near optimal for this step, scaling linearly with the number of available threads. Looking at the results for the Housing dataset in Figure 6.9 we can see that this is indeed the case for $P = 2$ and $P = 4$. The speedup is lower for $P = 8$, where the parallel implementation is only at most 5 times faster. Since our machine uses hyper-threading and only has 8 hyper-threaded CPUs, it can be expected that for $P = 8$ we do not get optimal performance for all 8 threads on the machine.

Figure 6.10 shows the parallel speedup for the other datasets. For the Retailer and USRetailer datasets, sorting for $P = 2$ is close to or even slower than the sequential sort of the root attributes. The reason for this is that the implemented parallel sorting algorithm applies a different sequential sort for $P = 1$, which in itself did not support parallelisation, but is fast for single-threaded execution. The multithreaded sorting algorithm shows expected speedups when P gets larger, but has difficulty surpassing the performance of the sequential sort for small P. This was less noticeable in the results of the Housing datasets, since the covers for Housing are nearly sorted by the cover-join. The aggregate step of the parallel algorithm does show a near-linear speedup for $P = 2$ and 4, except for the LastFM dataset. However, the cover of LastFM is small, so the overhead of parallelisation will be noticeable here.

For larger P, the experiments on USRetailer and Retailer are influenced more heavily by the garbage collector, where up to 40% of the time goes to garbage collection for $P = 8$ on the Retailer dataset, and up to 15% for $P = 8$ on USRetailer. It would be advised to rerun the parallelisation experiments for the Retailer dataset on a machine with more RAM.

All datasets split their work in almost equal amounts for the different threads, with a maximum size difference of up to 3% between different threads. Figure 6.11

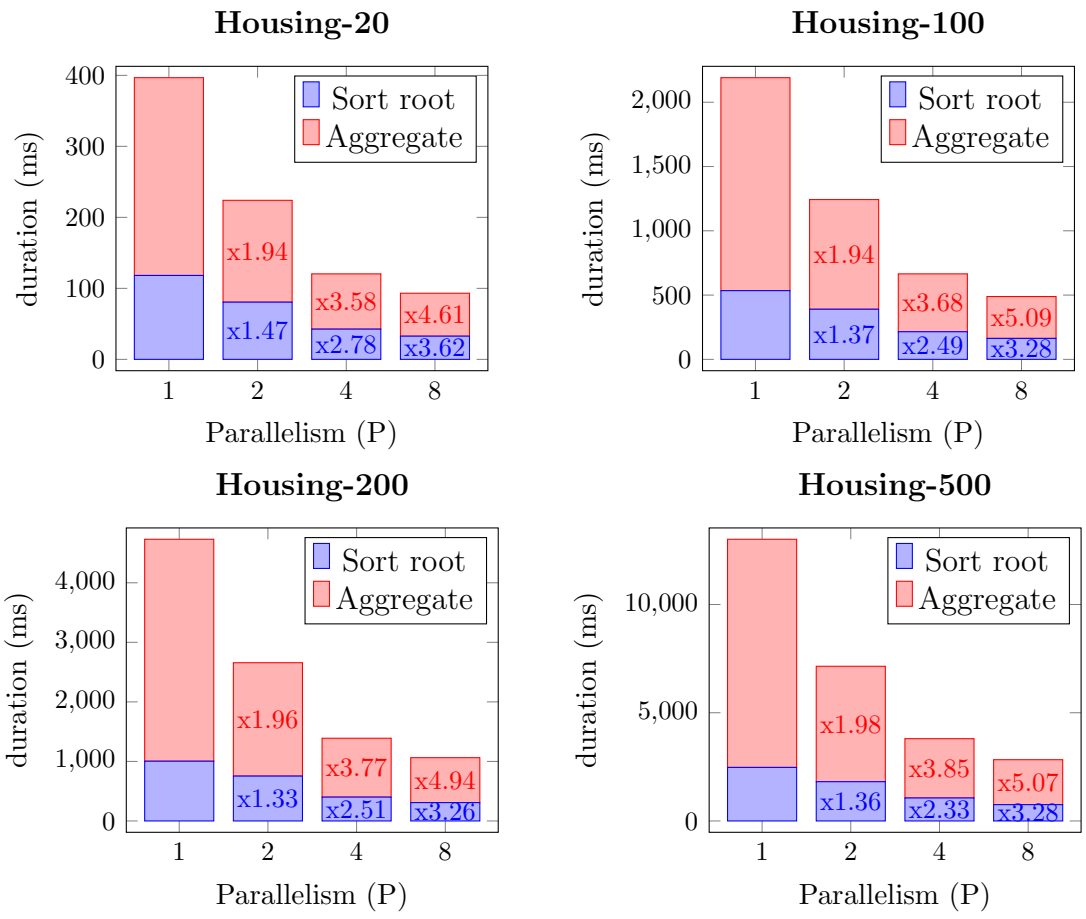


Figure 6.9: Parallelism speedups for HIER on the Housing datasets.
`SELECT SUM(crimesperyear) FROM R;`

shows traces of the different threads for the experiment on the RetailerSmall dataset with $P = 8$. Experiments on the other datasets have similar traces. With time going from left to right, first the hierarchical subquery H_{sku} is solved, followed by H_{day} and finally H_{root}. The horizontal red lines indicate that a thread is doing work. All three subqueries contain an "empty" block, which is where the Java library function `Arrays.parallelSort()` is called. This being a library call, we were not able to track when specific threads were doing work. Before and after the call to the `parallelSort` method are short phases of copying the block of data to sort and copying back the sorted block of data afterwards. This copying was necessary to convert it to an array of tuple instances such that it can be sorted with a custom comparator object through the library call. Various other (custom built) implementations for parallel sorting have been tested, but using the built-in Java sorting API gave the best results even including these copy steps. The final step for each subquery is to split the cover sorted on root attributes into P separate blocks and compute the aggregate. As seen by all threads starting and ending at similar times, the work is balanced fairly and near optimally.

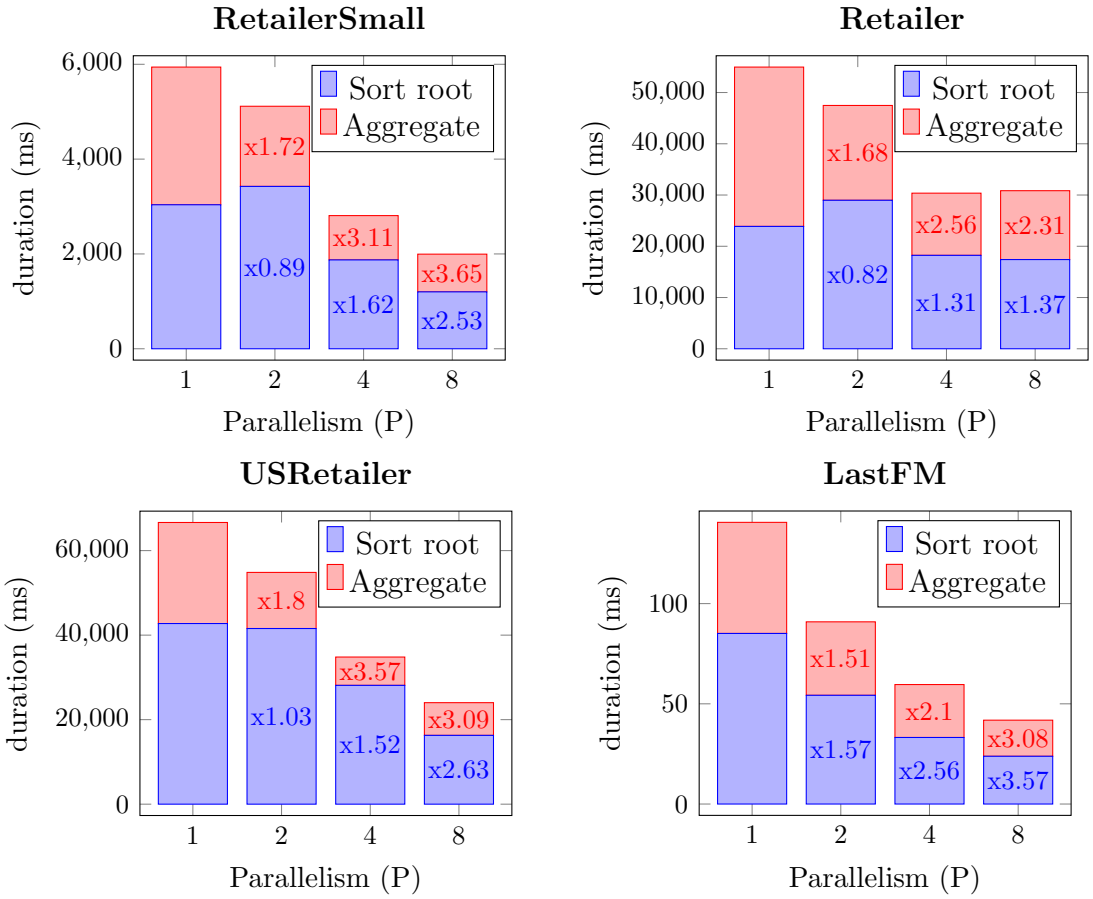


Figure 6.10: Parallelism speedups for HIER on the other datasets. Same `SUM` queries used as in Section 6.2

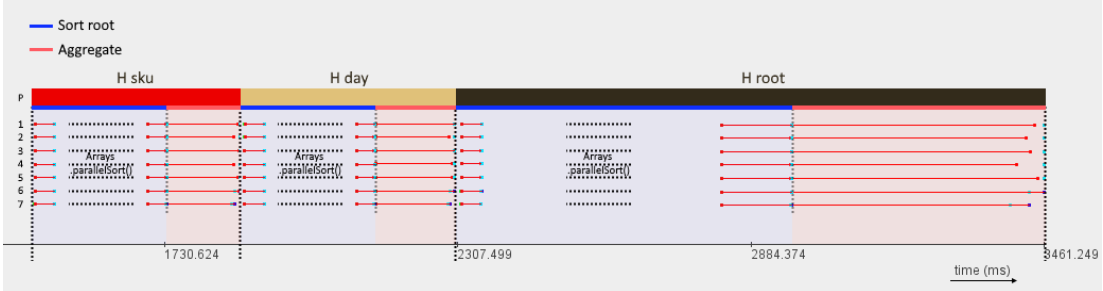


Figure 6.11: Parallelism division of work for RetailerSmall, $P = 8$

Chapter 7

Conclusions and Future Work

7.1 Conclusion

The objective of this thesis was to introduce new algorithms to solve aggregate queries over covers.

We showed how hierarchical schemas allow for efficient solving of aggregates over covers. With the use of the novel sorting technique called tree-sorting, it allows us to solve aggregate queries in a single pass over the cover and without extra memory. Extending this approach, it can be used to compute aggregates over arbitrary acyclic schemas using a novel decomposition into a graph of hierarchical sub-schemas.

The extensive benchmarks showed that the hierarchical solvers outperform existing solvers on hierarchical queries by at least 37%, and are up to several orders of magnitudes faster than the naive approach over a flat join result, closely following the gap in size between the cover relation and join result. For non-hierarchical acyclic join queries their performance suffers from sorting the several hierarchical sub-schemas, being close to or below the performance of competing solvers. We do show possible ideas to improve performance for solving non-hierarchical queries, which are encouraged to be explored.

Experiments of the described parallel implementation of the hierarchical solvers show good speed-ups, scaling almost linearly with the number of available threads for hierarchical queries. Non-hierarchical queries require more work on sorting the root attributes of each sub-schema in parallel, and therefore scale less well. The parallel sorting algorithm used in the implementation to sort a hierarchical schema on its root attributes can be improved, where in certain cases it was slower with two threads than the sequential run.

7.2 Future Work

There are various areas left to explore after the contributions of this thesis:

- We showed that repeated tree-sorting can be optimised by remembering the sort structure in a cover, such that consecutive calls to the tree-sorting algorithm can ignore parts of the variable order they sort on, as they would already be sorted by the original sort order. This has not been implemented in our algorithms, but is highly encouraged to make an attempt at incorporating this into the hierarchical solvers.

- Promising work has recently been presented on query compilation [10]. Generating low-level specialised bytecode for individual queries, query compilation allows us to optimise our implementations even further on a level that would not be possible or otherwise be unmaintainable using higher level languages. It is expected that because of the structure of the hierarchical solver, which operates over a tree of attributes, the current implementation suffers from the many function calls, as well as being too generic for the different types of paths to solve. There are many different factors to consider when solving a path in a hierarchical variable order: whether it needs to join with dependencies, should it export to a map or simply sum the aggregate, does it have any child paths to compute? Creating specialised solvers for every possible combination of factors is unreasonable, but with the help of code compilation one can easily and in a maintainable way compile specialised code for each path.

- The experiments on non-hierarchical schemas gave the hierarchical decomposition as input to the solvers. These decompositions have been hand-crafted to be as small as possible, it would be interesting to find out how one can select the "best" hierarchical decomposition, and whether one can say anything about the minimum size of a hierarchical decomposition for a given query.

- Some improvements to the parallel implementation of the hierarchical solver can be made. For instance, the solver does not divide its work fairly if there are only a handful of different values for the root attributes of the hierarchical schema to solve on. An improvement would be to recursively split work for the same root values across different threads and merge together the results of each thread afterwards. Additionally, the parallel sorting algorithm used to sort the root attributes of a hierarchical schema in the parallelised hierarchical solver

can be improved to scale better in performance when increasing the number of threads. Our experiments contained samples where sorting with two threads was slower than executing the single-threaded algorithm.

- Extend the hierarchical solvers to other aggregate queries, such as group-by queries. Additionally, one could look at supporting selection, projection and ordering with our hierarchical solvers, or develop a new suite of algorithms for these queries such that a full query engine can be constructed that operates over covers.

Appendix A

Datasets

Further details about the different datasets used in the experiments are given here. The schemas of the separate tables in each dataset are given, as well as the composition of these tables (size, number of non-dangling tuples), size of its corresponding cover generated by the cover-join algorithm in [6] with implementation in Algorithm C.10 in Appendix C, and hierarchical decomposition used in the HIER solver.

A.1 Housing

The Housing dataset consists of the following tables:

House
postcode
livingarea
price
nbbedrooms
nbbathrooms
kitchensize
house
flat
condo
garden
parking

Shop
postcode
openinghoursshop
pricerangeshop
sainsburys
tesco
ms

Institution
postcode
typeeducation
sizeinstitution

Restaurant
postcode
openinghoursrest
pricerangerest

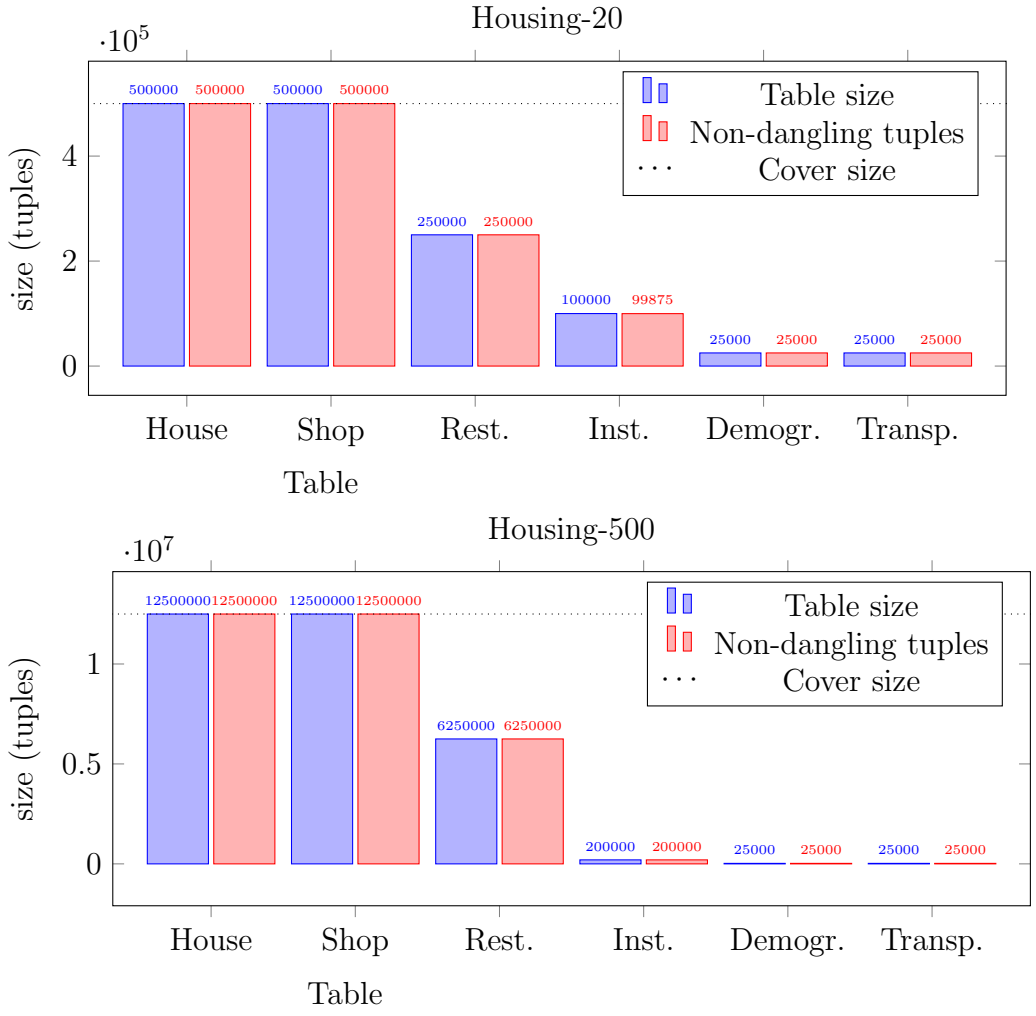
Demographics
postcode
averagesalary
crimesperyear
unemployment
nbhospitals

Transport
postcode
nbbuslines
nbtrainstations
distancecitycentre

The natural join of these tables is a star-join on the `postcode` attribute, and is a hierarchical query. The individual tables scale roughly as follows, where n is the dataset:

Table	Size (tuples)
House	$25000 * n$
Shop	$25000 * n$
Institution	$25000 * \log n$
Restaurant	$25000 * n/2$
Demographics	25000
Transport	25000

The full join contains roughly $12500 * n^3 \log n$ tuples, the cover for dataset n contains $25000 * n$ tuples. The figures below show the individual sizes of the tables in the join, the amount of non-dangling tuples in these tables, and the number of tuples in the cover relation.



Hierarchical Decomposition Since the natural join for the Housing dataset is hierarchical it is the only subquery of the hierarchical decomposition.

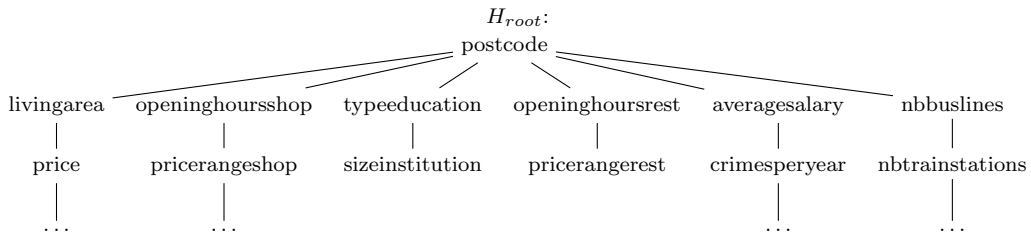


Figure A.1: Hierarchical decomposition for Housing with 1 hierarchical subquery.

A.2 LastFM

The LastFM dataset consists of the following tables:

Usertaggedartiststimestamps
userid
artistid
tagid
timestamp

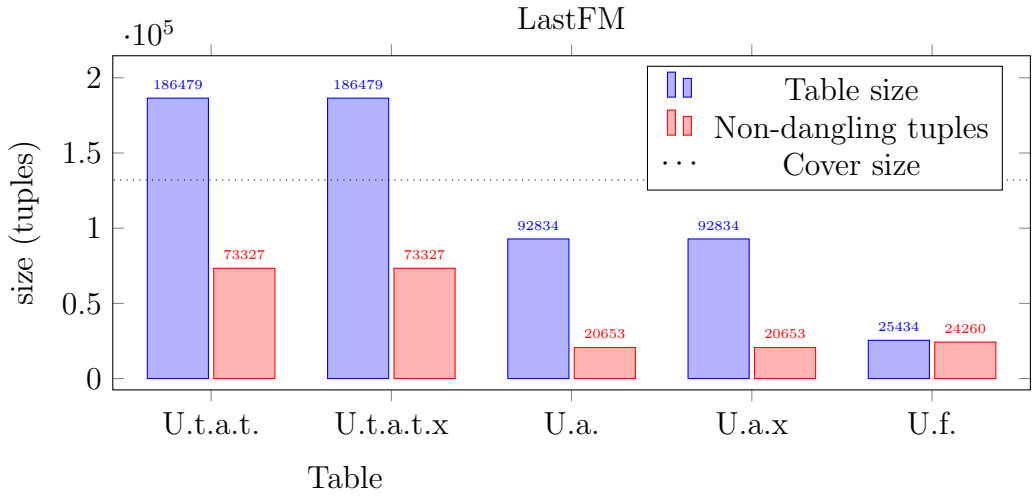
Userartists
userid
artistid
weight

Userfriends
userid
friendid

Usertaggedartiststimestampsx
friendid
artistidx
tagidx
timestampx

Userartistsx
friendid
artistidx
weightx

The tables in LastFM contain a high amount of dangling tuples, as seen in the figure below:



Hierarchical Decomposition The smallest hierarchical decomposition for the LastFM dataset consists of two hierarchical subqueries, as the natural join is not hierarchical in itself.

Figure A.2: Hierarchical decomposition for LastFM with 2 hierarchical subqueries.

A.3 Retailer

The RetailerSmall and Retailer datasets consists of the following tables:

promotedF
sku
store
day
promoInd

dayOfWeek
day
dow

monthNumber
day
moy

dayToQuarter
day
quarter

dowOccurrenceInMonth
day
dowOcc

countyF
store
county

tvRegionLabelF
store
tvRegion

townF
store
town

store_to_countryF
store
country

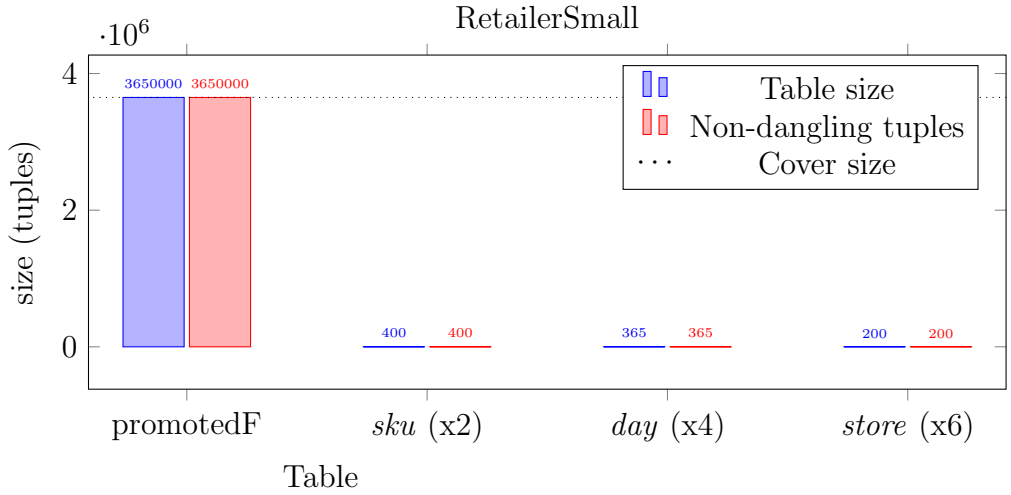
store_to_regionF
store
region

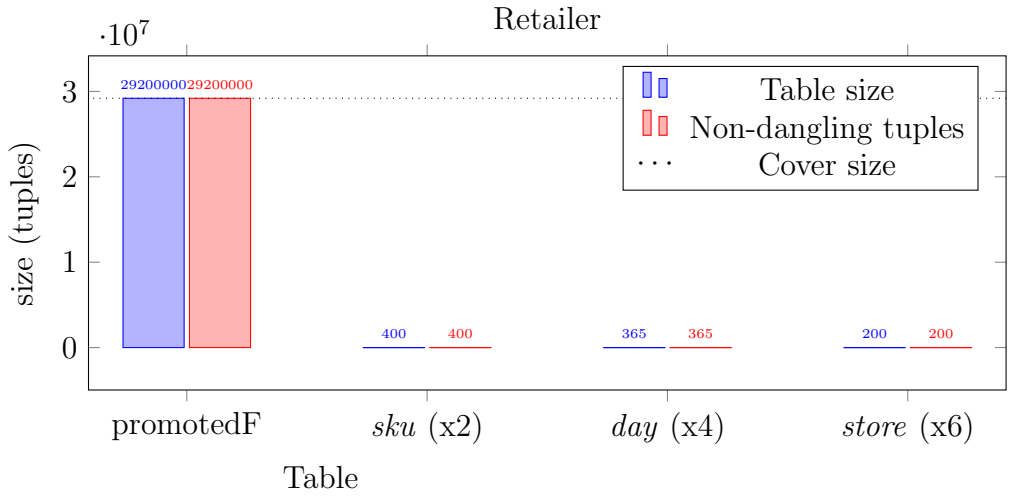
locationTypeF
store
loctype

sku_to_subSection
sku
subsection

sku_to_section
sku
section

The difference between RetailerSmall and Retailer is the size of the *promotedF* table, which contains 3,650,000 tuples in RetailerSmall and 29,200,000 tuples in Retailer.





Hierarchical Decomposition The hierarchical decomposition used for the Retailer dataset consists out of three hierarchical subqueries. There are multiple decompositions possible, but we have opted for this specific decomposition as it has the largest possible root subquery.

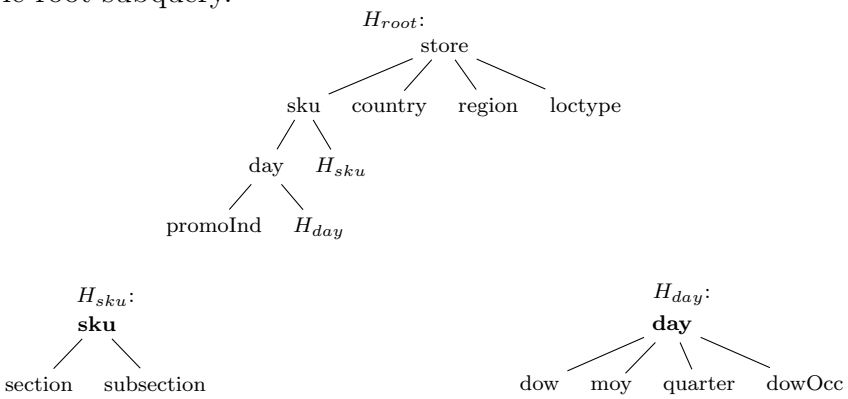


Figure A.3: Hierarchical decomposition for Retailer with 3 hierarchical subqueries.

A.4 USRetailer

Census
zip
population
white
asian
pacific
black
medianage
occupiedhouseunits
houseunits
families
households
husbwife
males
females
householdschildren
hispanic

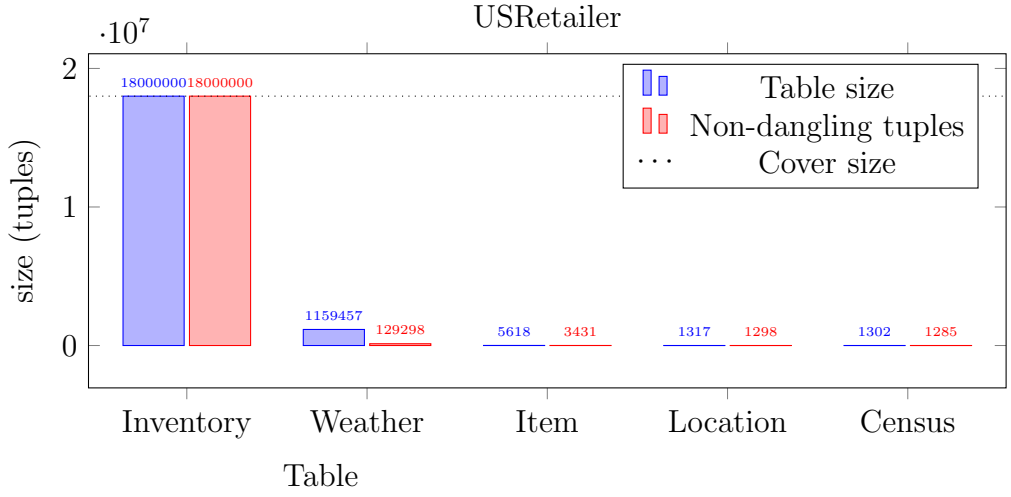
Location
locn
zip
rgn_cd
clim_zn_nbr
tot_area_sq_ft
sell_area_sq_ft
avghhi
supertargetdistance
supertargetdrivetime
targetdistance
targetdrivetime
walmartdistance
walmartdrivetime
walmartsupercenterdistance
walmartsupercenterdrivetime

Item
ksn
subcategory
category
categoryCluster
prize

Inventory
locn
dateid
ksn
inventoryunits

Weather
locn
dateid
rain
snow
maxtemp
mintemp
meanwind
thunder

USRetailer consists of one large *Inventory* table joined with a number of smaller tables. These smaller tables consist of a number of dangling tuples:



Hierarchical Decomposition The USRetailer dataset must have at least three hierarchical subqueries in its hierarchical decomposition. Choosing the decomposition with the largest root subquery gives the following:

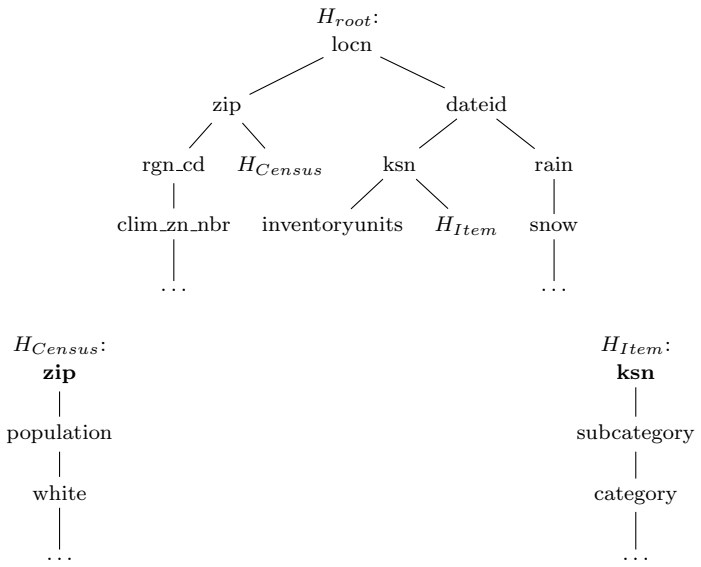


Figure A.4: Hierarchical decomposition for USRetailer with 3 hierarchical subqueries.

Appendix B

Experiment Setup

All experiments were executed on Google Cloud Compute Engine[1], running on machine type `n1-highmem-8`[2]. The exact specifications of this machine and versions of the software used are as follows:

- Intel(R) Xeon(R) Scalable Processor (Skylake) 2.00GHz
- 64-bit architecture
- 8 vCPUs
- 52GB RAM
- Linux SMP Debian 4.9.110-3+deb9u2 (2018-08-13) x86_64 GNU/Linux
- OpenJDK 1.8.0_181
- Kotlin 1.2.10

Care must be taken when benchmarking on the Java Virtual Machine. Best effort has been made to produce accurate results with our benchmarks, following the guidance in an excellent article by Oracle[3]. Each experiment is executed separately in their own JVM instance, and the JVM is warmed up by running each experiment at least 3 times and for a minimum of 10 seconds. Afterwards, each experiment is executed for a number of runs depending on the duration of the experiment, but for at least 20 seconds and 4 runs. The average wall-clock time of these runs is reported

[1] `https://cloud.google.com/compute/`

[2] `https://cloud.google.com/compute/docs/machine-types`

[3] `http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html`

in the results, retrieved using Java's `System.nanoTime()`. The time required to load the datasets into memory is ignored.

Before each run of an experiment starts, we call the Garbage Collector several times through `System.gc()`, and ignore the time it takes to perform the garbage collection. We also log garbage collection events to detect whether our results have been influenced by a "stop-the-world" event of the garbage collector. Each experiment is run with the flags `-Xmx48G -Xms48G`, which increase the initial and max-heap size to 48GB. This prevents most of the garbage collection calls during timed runs in the experiments, as the heap immediately starts off at its maximum size. Garbage collection is triggered when the heap is full, so starting with a smaller initial heap will cause repeated garbage collection and resizing of the heap until it reaches its maximum size.

Most experiments did not see any garbage collection events while timing a run, except for the largest datasets Retailer and USRetailer. Chapter 6 goes into more detail about the influence of garbage collection for each dataset.

Appendix C

Tools

C.1 Cover-Join Implementation

The proposed algorithm for the Cover-Join operator in Chapter 3 is implemented in Algorithm C.10. To compute $R_1 \mathbin{\mathring{\bowtie}} R_2$, first group tuples for R_1 and R_2 on their values for the join attributes $J = S(R_1) \cap S(R_2)$, while preserving the relative ordering for tuples with the same join values. This is important to keep any sorted structure in place per block of tuples with equal join values. Next, for each join key j in ascending order, add the minimum edge cover over tuples in R_1 and R_2 having join key j to the resulting cover relation.

Algorithm C.10 Implementation of Cover-Join operator

```
1:  procedure CoverJoin(R_1, R_2)                          ▷ Cover-join R_1 with R_2.
2:      J ← S(R_1) ∩ S(R_2)                                ▷ join attributes
3:      tuples_1 ← {}                                      ▷ mapping from join value j to tuples in R_1 with j
4:      tuples_2 ← {}                                      ▷ mapping from join value j to tuples in R_2 with j
5:      for each tuple t ∈ R_1 do
6:          tuples_1[π_J t] += t
7:      for each tuple t ∈ R_2 do
8:          tuples_2[π_J t] += t
9:      R_result ← []                                      ▷ resulting cover relation, initially empty
10:     J_sorted ← keys(tuples_1) sorted in ascending order
11:     for each join key j ∈ J do
12:         if j not in tuples_1 or j not in tuples_2 then
13:             ▷ One of the relations does not have tuples with join value j, skip j
14:             continue
15:         join_1 ← tuples_1[j]
16:         join_2 ← tuples_2[j]
17:         for i in 0 until max(|join_1|, |join_2|) do
18:             t_1 ← join_1[min(i, |join_1| − 1)]
19:             t_2 ← join_2[min(i, |join_2| − 1)]
20:             add t_1 joined with t_2 to R_shuf
21:     return R_result
```

C.2 Create Shuffled Cover

It is straightforward to shuffle a regular relation: you randomly reorder its tuples. Shuffling a cover is more complex, only shuffling the rows of the cover relation is not enough, as shuffling the complete rows will preserve parts of the sorted structure deeper in the sort tree. Only the root attributes of the tree have been shuffled now, but when these are sorted back into place the remaining part of the sort tree is immediately sorted. To make sure that the cover is shuffled on all levels of the sort tree a custom cover-join implementation is given that constructs a shuffled cover. Applying this shuffled cover-join algorithm similarly to the normal cover-join over a Cover-Join Plan creates a cover that is shuffled correctly. Algorithm C.11 shows this shuffled cover-join implementation.

Algorithm C.11 Create shuffled cover

```
1:  procedure ShuffledCoverJoin(R_1, R_2)        ▷ Cover-join R_1 with R_2, such that the result is shuffled.
2:      J ← S(R_1) ∩ S(R_2)                                                          ▷ join attributes
3:      tuples_1 ← {}                                  ▷ mapping from join value j to tuples in R_1 with j
4:      tuples_2 ← {}                                  ▷ mapping from join value j to tuples in R_2 with j
5:      for each tuple t ∈ R_1 do
6:          tuples_1[π_J t] += t
7:      for each tuple t ∈ R_2 do
8:          tuples_2[π_J t] += t
9:      R_shuf ← []                                                  ▷ resulting cover relation, initially empty
10:     for each join key j ∈ J do
11:         if j not in tuples_1 or j not in tuples_2 then
12:             ▷ One of the relations does not have tuples with join value j, skip j
13:             continue
14:         join_1 ← tuples_1[j]
15:         join_2 ← tuples_2[j]
16:         shuffle join_1 and join_2
17:         for i in 0 until max(|join_1|, |join_2|) do
18:             t_1 ← join_1[min(i, |join_1| − 1)]
19:             t_2 ← join_2[min(i, |join_2| − 1)]
20:             add t_1 joined with t_2 to R_shuf
21:     shuffle tuples in R_shuf
22:     return R_shuf
```

Bibliography

[1] N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *Proc. VLDB Endow.*, 6(14):1990–2001, Sept. 2013.

[2] N. Bakibayev, D. Olteanu, and J. Závodný. Fdb: A query engine for factorised relational databases. *Proc. VLDB Endow.*, 5(11):1232–1243, July 2012.

[3] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[4] Fdb research. https://fdbresearch.github.io/.

[5] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[6] A. Kara and D. Olteanu. Covers of query results. *CoRR*, abs/1709.01600, 2017.

[7] H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database factorized learning. In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017.*, 2017.

[8] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, Mar. 2015.

[9] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 3–18, New York, NY, USA, 2016. ACM.

[10] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *Proceedings of the 2016 International*

Conference on Management of Data, SIGMOD '16, pages 1907–1922, New York, NY, USA, 2016. ACM.

[11] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[12] B. A. M. van Geffen. Qr decomposition of normalised relational data.

[13] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 345–357, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.