Scalable Query Evaluation in Relational Databases

Rasmus Resen Amossen

PhD dissertation, November 30, 2010

Supervisor: Rasmus Pagh
IT University of Copenhagen, Denmark

Assessment committee: Thore Husfeldt (chair)
IT University of Copenhagen, Denmark

Peter Boncz
Centrum Wiskunde & Informatica, Netherlands

Rolf Fagerberg
University of Southern Denmark, Denmark

The *Efficient Computation* research group, IT University of Copenhagen, Denmark, 2010

Abstract

The scalability of a query depends on the amount of data that needs to be accessed when computing the answer. This implies three immediate general strategies for improving query performance: decrease the amount of data (including intermediate results) to be accessed by accessing it smarter; decrease the amount by simply reducing the data quantity in the first place; and increase the amount of data accessed per time unit. This PhD dissertation presents four research results, covering each of these three approaches.

The first three results focus on variations of the highly applicable query class *join-project*, which is a join of two database tables followed by a duplicate eliminating projection. Join-projects are equivalent to sparse Boolean matrix multiplication and frequent pair mining (the special case of frequent itemset with itemset cardinality limited to 2).

We describe a new output sensitive algorithm for join-projects which has small intermediate results on worst-case inputs, and in particular, is efficient in both the RAM and I/O model. The algorithm uses the output size to deduce its computation strategy, and this introduces a chicken-and-egg problem: how do we obtain the output size without actually computing the output? This question is answered in another result in which we obtain a $(1 \pm \varepsilon)$ approximation of the output size in expected linear time and I/O for $\varepsilon > 1/\sqrt[4]{n}$.

In another result we address the throughput itself by using the massive parallel capabilities of graphics processing units (GPUs) to handle the pair mining problem. For that we present a new data structure, BatMap, which is a novel vertical data layout that is particularly well suited for parallel processing.

The last result deals with the general problem of reducing the quantity of data that must be accessed for answering any given query on a row store RDBMS. We present a quadratic integer program formulation of the vertical partitioning problem for OLTP workloads in a distributed environment. This quadratic optimization problem is NP-hard so we also describe a randomized heuristic that empirically has shown to be reliable in sense of both speed and cost reduction.

Preface

This text constitutes my dissertation for the PhD degree in computer science at the IT University of Copenhagen. It is the result of work done from 2007 to 2010 under supervision of Rasmus Pagh.

I would like to express my gratitude to my advisor, Rasmus Pagh, for his interest, support, and insightful discussions throughout the years. I am also pleased to thank Daniel Abadi for letting me visit his department at Yale for two months in 2009. Further, I would like to thank my wife, Line, and my two sons, Elias and Noah, for their patience and understanding during occasionally stressful periods preceding article submission deadlines.

Rasmus Resen Amossen
Copenhagen, November 2010

Contents

Chapter 1

Introduction

In 2006, Pagh and Pagh [74] presented an algorithm for computing an acyclic join of k database relations in a way that scales well in k. Their result led to two hypothesises:

Hypothesis 1: Queries that are a combination of an acyclic join and a duplicate eliminating projection can be evaluated in a way that scales to a large number of relations.

Hypothesis 2: Join queries whose join graph has constant size and contains a single cycle can be evaluated in a scalable way in the sense that internal memory size and external memory speed need to grow only linearly with the data size.

It was originally a goal to initialize this PhD project with an investigation of these two hypotheses. However, it quickly became clear, that computing an acyclic join followed by a duplicate eliminating projection (in the following denoted a *join-project*) is equivalent to computing a sparse Boolean matrix multiplication—a problem that has been studied intensively for decades. Also, we can compute a join-project of k relations as a sequence of matrix multiplications. As we shall see, Boolean matrix multiplication is unlikely to scale well in k, so Hypothesis 1 is unlikely to hold as well. The smallest possible join graph containing a single cycle is a simple three-cycle. Unfortunately, this simple example can also be reduced to the problem of computing a Boolean matrix multiplication, and the problem becomes increasingly complex for join graphs with larger cycles.

With Hypothesis 2 being more complicated than Hypothesis 1, the focus therefore changed from investigation of the two hypotheses to a broader research on join-project related algorithms and data structures, and luckily we were able to obtain some results. First, we presented an *output sensitive* algorithm for join-projects, in the sense that the run time is defined by the output size. It might sound counter intuitive to require knowledge of the size of the output before it is actually computed, but nonetheless we presented

an algorithm for efficiently estimating this size with high accuracy and high confidence. In a third result, we focued on a way to compute join-projects efficiently using modern GPUs as computation devices, and we presented a new data layout that supports a high degree of parallel processing as required by the parallel nature of GPUs.

Simultanously with the research in join-projects, I followed, with big interest, another research project on a paradigm shifting database system: H-store [91]. In 2009 I had the privilege of visiting Daniel Abadi (a co-researcher in the H-store project) and his database research group at the Yale university in New Haven, CT, USA. During the stay at Yale, I worked with partitioning strategies for H-Store, and this led to a paper on a model and a heuristic for vertical partitioning in databases with an H-store like architecture.

1.1 Contributions

The contributions of this PhD are collected in four papers:

Chapter 3 is an extension of the paper *Better Size Estimation for Sparse Matrix Products* [9] presented at the APPROX/RANDOM 2010 conference in Barcelona, Spain. The paper was written in collaboration with Rasmus Pagh and Andrea Campagna.

Chapter 4 elaborates on the paper *Faster Join-Projects and Sparse Matrix Multiplications* [8] presented at the ICDT 2009 conference in St. Petersburg, Russia. It was written in collaboration with Rasmus Pagh.

Chapter 5 elaborates on the paper *A New Data Layout For Set Intersection on GPUs*. It was written in 2010 in collaboration with Rasmus Pagh.

Chapter 7 contains the paper *Vertical partitioning of relational OLTP databases using integer programming* [7] presented at the 5th International Workshop on Self Managing Database Systems (SMDB 2010), Long Beach, California, USA.

1.2 Structure

The remainder of the text is structured as follows. Part I covers everything related to the research on join-projects, and Part II contains a single chapter about the result on vertical partitioning. Part I furthermore contains an introduction giving a general overview of join-projects and related problems. Following this are three papers that each elaborates on a research paper, and Part I concludes with a chapter that aims to draw some broader perspectives to the topics addressed in the papers.

Part I

Finding connected pairs

Chapter 2

Overview

The fundamental and relevant problem og finding pairs of related entities in datasets often occurs in multiple disguises and has therefore frequently been addressed in algorithm and data mining research. Below we briefly describe four variations of the problem.

Sparse Boolean matrix multiplication: Consider two $n \times n$ sparse Boolean matrices M' and M''. We are interested in the Boolean product $M'M''$ where, for row r and column c, we have $(M'M'')_{r,c} = 1$ if $\sum_{i \leq n} M'_{r,i} M''_{i,c} > 0$. As described below, this product can be computed in less than $\mathcal{O}(n^3)$ time.

Join-project: Consider two database tables in the general form $R_1 = (a, b)$ and $R_2 = (b, c)$ sharing a common attribute b, and consider a join of the two tables followed by a duplicate eliminating projection that projects away the join attribute. In relational algebra this operation is written $\pi_{a,c}(R_1 \bowtie R_2)$. To see that this problem is equivalent to that of Boolean matrix multiplication, represent each of R_1 and R_2 as affinity matrices, connecting b values to a and c values, respectively. Section 4.3 on page 34 explains this in further detail.

Frequent itemset: In the frequent itemset problem we are given a set of transactions $T_1, \ldots, T_m \subseteq \{1, \ldots, n\}$. Each transaction holds one or more of n possible items, and the challenge is now to find the subsets S where $S \subseteq T_i$ for at least two i. S is then said to have *support* at least two among the transactions. If we add the restriction that $|S| = 2$ (i.e. only item *pairs* are considered), the problem becomes equivalent to sparse Boolean matrix multiplication. As with join-projects, this can be seen by using an affinity matrix to represent the relationship between transactions and items.

Relating pairs: The general problem of relating pairs can often be reduced to sparse Boolean matrix multiplication if grouping of pairs is based

on equality of a common property. Examples include finding friends of friends in social networks, actors playing together in at least one movie, and students that have ever attended a course together.

In the following chapters we approach this class of problems from three perspectives. Chapter 3 does not deal with the computation of the result itself, but instead describes how to compute an *estimate* of its size. This is relevant in multiple situations: first of all, as the computation of the output can be very time and space consuming, it may be desirable to know in advance if the problem instanse in question is practically solvable on the available hardware; secondly, the output size can also be used as a parameter for matrix multplication algorithms (we present an example of such an algorithm in Chapter 4); last, for matrix multiplications involving more than two matrices, the optimal multiplication order strongly depends on the size of the intermediate sub-products.

Our second approach to pair generation, described in Chapter 4, is an output sensitive algorithm for sparse Boolean matrix multiplication. The algorithm can be thought of as a hybrid between classical pair generation, using a *merge-join*, and any matrix multiplication algorithm. Given the expected output size, it divides the input into two classes, depending on density, and forms the final output by combining the results from a merge-join of the sparse class, and a matrix multiplication of the dense class.

In Chapter 5 we explore the simple, yet powerful, architecture of graphics processing units (GPUs) supporting massive parallelism. We present a new space efficient data structure called BatMaps for representing and intersecting sets. With this data structure we exploit the parallel capabilities of GPUs, and we prove that the data structure is actually scalable in practice—this time from the perspective of the frequent itemset problem.

Chapter 3

Size estimation

This chapter is an extension of the paper *Better Size Estimation for Sparse Matrix Products* [9] presented at the APPROX/RANDOM 2010 conference in Barcelona, Spain. The paper was written in collaboration with Rasmus Pagh and Andrea Campagna. Some of the text relies heavily on a few elements from probability theory, so in order to ease reading, we will briefly mention these below. More elaborate explanations can be found in [34, 64].

The *mean* or *expected value* (not to be confused with the *most probable* value) of a discrete random variable X is denoted $\mathbf{E}[X]$, and is simply the weighted sum of all possible values in the relevant universe U and their probabilities, that is: $\mathbf{E}[X] = \sum_{x \in U} x\mathbf{Pr}[X = x]$.

Theorem 3.1 *The expected value operator* $\mathbf{E}[\cdot]$ *is linear.*

For a random variable X we use the *variance* $\mathbf{Var}(X)$ to describe how far values in U lie from their mean. By definition, the variance is given as follows:

Definition 3.2 (Variance) $\mathbf{Var}(X) = \mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2] - \mathbf{E}[X]^2$.

The estimation method to be presented is a randomized algorithm, implying that two consecutive computations are likely to produce different estimates. Therefore, we will need a couple of tools to bound the error probability. The first tool, *Chebychev's inequality*, states, that in any data sample or probability distribution, the majority of the values are close to the mean value when the variance is small. More formally:

Theorem 3.3 (Chebyshev's inequality) *For any* $\alpha > 0$,

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq \alpha] \leq \frac{\mathbf{Var}(X)}{\alpha^2}.$$

In our specific case, we will use the inequality to show that the core estimation algorithm guarantees a result within a factor $(1 \pm \varepsilon)$ from the real size with error probability $\delta = 1/3$.

We would like to decrease this error probability arbitrarily. This can be done by repeating the experiments and utilizing a *Chernoff bound* which bounds the probability that a majority of independent discrete indicator random variables (Bernoulli random variables) take a given value.

Theorem 3.4 (Chernoff bound) *Let $X_1, \ldots, X_k$ be independent random Bernoulli variables and let $X = \sum_i X_i$. Also, let μ denote $\mathbf{E}[X]$. Then, for any $\alpha > 0$,*

$$\mathbf{Pr}[X < (1-\alpha)\mu] < \left(\frac{e^{-\alpha}}{(1-\alpha)^{(1-\alpha)}}\right)^{\mu} < e^{-\mu\alpha^2/2}.$$

In our case we will have k Bernoulli random variables, X_i, corresponding to k estimates where $X_i = 1$ if estimate i is a $(1 \pm \varepsilon)$ estimate and $X_i = 0$ otherwise. As $\mathbf{E}[X] = \mu = 2k/3$ we can ask for the probability that at least half of the estimates are good by setting $\alpha = 1/4$:

$$\mathbf{Pr}[X < (1 - 1/4)2k/3] = \mathbf{Pr}[X < k/2] < e^{-k/48}.$$

In other words, the probability δ of getting fewer than $k/2$ estimates within $(1 \pm \varepsilon)$ of the actual value decreases exponentially with the number of experiments k. Consequently, the number k of experiments needed to achieve any given error probability δ is in the order of $\mathcal{O}(\log(1/\delta))$, and knowing that a majority of the estimates are within the desired bound, we can report the estimate found at the median.

Last, the chapter also makes use of *pairwise independence* of families of hash functions. This means, that the distribution of the image of any randomly chosen function from this family can be considered random. More formally:

Definition 3.5 (Pairwise independence) *Two random variables X and Y in a universe U are* pairwise independent *if $\mathbf{Pr}[X \in x] = \mathbf{Pr}[X \in x | Y \in y]$ for any $x, y \subseteq U$. A hash function $h : D \to I$ for any domain D and image I is pairwise independent, if for any two $a, b \in D$ and $i, j \subseteq I$, it satisfies*

$$\mathbf{Pr}[h(a) \in i] = \mathbf{Pr}[h(a) \in i | h(b) \in j].$$

This definition implies that, if X and Y are pairwise independent, then

$$\mathbf{Pr}[X \in x \wedge Y \in y] = \mathbf{Pr}[X \in x]\mathbf{Pr}[Y \in y]$$

for $x, y \in U$.

Lemma 3.6 *Let $h_1, h_2 : U \to [0; 1]$ be pairwise independent hash functions, and define $h : U \times U \to [0; 1]$ by $(x, y) \mapsto (h_1(x) - h_2(y)) \mod 1$. Then h is also pairwise independent.*

PROOF First, let $X = h(a, b_1)$ and $Y = h(a, b_2)$ for $a, b_1, b_2 \in U$. Then for any $x, y \subseteq [0; 1]$ we have

$$\mathbf{Pr}[X \in x \wedge Y \in y] = \\ \mathbf{Pr}[(h_1(a) - h_2(b_1)) \mod 1 \in x \wedge (h_1(a) - h_2(b_2)) \mod 1 \in y].$$

$h_1(a)$ can be treated as a constant $c \in [0; 1]$ and since h_2 is pairwise independent we have

$$\mathbf{Pr}[(h_1(a) - h_2(b_1)) \mod 1 \in x \wedge (h_1(a) - h_2(b_2)) \mod 1 \in y] = \\ \mathbf{Pr}[(c - h_2(b_1)) \mod 1 \in x]\mathbf{Pr}[(c - h_2(b_2)) \mod 1 \in y]$$

which shows the pairwise independence for this case. Similar arguments can be applied for $X = h(a_1, b)$ and $Y = h(a_2, b)$ as well as $X = h(a_1, b_1)$ and $Y = h(a_2, b_2)$. □

Lemma 3.7 *If $Y = X_1 + \cdots + X_n$ is a sum of pairwise independent Bernoulli variables, then $\mathbf{Var}(Y) \leq \mathbf{E}[Y]$.*

We use the same techniques as in [84] to prove the lemma.

PROOF As X_i and X_j are pairwise independent, $\mathbf{E}[X_i X_j] = \mathbf{E}[X_i]\mathbf{E}[X_j]$. Thus,

$$\begin{aligned} \mathbf{E}[Y^2] &= \mathbf{E}[(X_1 + \cdots + X_n)^2] \\ &= \mathbf{E}\left[\sum_{i=1}^{n} X_i^2\right] + \mathbf{E}\left[\sum_{i \neq j} X_i X_j\right] \\ &= \sum_{i=1}^{n} \mathbf{E}[X_i^2] + \sum_{i \neq j} \mathbf{E}[X_i X_j] \\ &= \sum_{i=1}^{n} \mathbf{E}[X_i^2] + \sum_{i \neq j} \mathbf{E}[X_i]\mathbf{E}[X_j]. \end{aligned}$$

Also

$$\mathbf{E}[Y]^2 = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right]^2 = \left(\sum_{i=1}^{n} \mathbf{E}[X_i]\right)^2 = \sum_{i=1}^{n} \mathbf{E}[X_i]^2 + \sum_{i \neq j} \mathbf{E}[X_i]\mathbf{E}[X_j].$$

Therefore,

$$\begin{aligned}\mathbf{Var}(Y) &= \mathbf{E}[Y^2] - \mathbf{E}[Y]^2 \\ &= \sum_{i=1}^{n} \mathbf{E}[X_i^2] + \sum_{i \neq j} \mathbf{E}[X_i]\mathbf{E}[X_j] - \left(\sum_{i=1}^{n} \mathbf{E}[X_i]^2 + \sum_{i \neq j} \mathbf{E}[X_i]\mathbf{E}[X_j] \right) \\ &= \sum_{i=1}^{n} \mathbf{E}[X_i^2] - \sum_{i=1}^{n} \mathbf{E}[X_i]^2 = \sum_{i=1}^{n} \mathbf{E}[X_i] - \sum_{i=1}^{n} \mathbf{E}[X_i]^2 \\ &\leq \sum_{i=1}^{n} \mathbf{E}[X_i] = \mathbf{E}[Y].\end{aligned}$$

□

3.1 Introduction

In this chapter we will consider a $d \times d$ Boolean matrix as the subset of $\{1, \ldots, d\} \times \{1, \ldots, d\}$ corresponding to the nonzero entries. The product of two matrices R_1 and R_2 contains (i, k) if and only if there exists j such that $(i, j) \in R_1$ and $(j, k) \in R_2$. The matrix product can also be expressed using basic operators of relational algebra: $R_1 \bowtie R_2$ denotes the set of tuples (i, j, k) where $(i, j) \in R_1$ and $(j, k) \in R_2$, and the projection operator π can be used to compute the tuples (i, k) where there exists a tuple of the form $(i, \cdot, k)$ in $R_1 \bowtie R_2$. Since most of our applications are in database systems we will primarily use the notation of relational algebra.

We consider the following question: given relations R_1 and R_2 with schemas (a, b) and (b, c), estimate the number of *distinct* tuples in the relation $Z = \pi_{ac}(R_1 \bowtie R_2)$. This problem has been referred to in the literature as *join-project* or *join-distinct*[1]. We define $n_1 = |R_1|$, $n_2 = |R_2|$, and $n = n_1 + n_2$. As observed above, the join-project problem is equivalent to the problem of estimating the number of non-zero entries in the product of two Boolean matrices, having n_1 and n_2 non-zero entries, respectively.

In recent years there has been several papers presenting new algorithms for sparse matrix multiplication [8, 60, 104]. In particular, these algorithms can be used to implement Boolean matrix multiplication. However, even if matrix multiplication could be done in quadratic time which is the lower bound conjectured by many, the proposed algorithms all have substantially superlinear time complexity in the input size n: on worst-case inputs they

[1]Readers familiar with the database literature may notice that we consider projections that return a set, i.e., that projection is duplicate eliminating. We also observe that any equi-join followed by a projection can be reduced to the case above, having two variables in each relation and projecting away the single join attribute. Thus, there is no loss of generality in considering this minimal case.

require time $\omega(n^{4/3})$, even when the output Z only has size $\mathcal{O}(n)$. Observation 4.7 on page 38 will show an example of this based on an algorithm for matrix multiplication that we will develop in Chapter 4.

In an influential work, Cohen [22] presented an estimation algorithm that, for any constant error probability $\delta > 0$, and any $\varepsilon > 0$, can compute a $1 \pm \varepsilon$ approximation of $z = |Z|$ in time $\mathcal{O}(n/\varepsilon^2)$. Cohen's algorithm applies to the more general problem of computing the size of the transitive closure of a graph, which an extension of the edge set so that we add an edge from u to v if there exists a path from u to v.

Our main result is that in the special case of sparse matrix product size estimation, we can improve this to expected time $\mathcal{O}(n)$ for $\varepsilon > 4/\sqrt[4]{n}$. This means that we have a linear time algorithm for relative error where Cohen's algorithm would use time $\mathcal{O}(n^{3/2})$.

Approach. To build intuition on the size estimation question, consider the sets $\mathcal{A}_j = \{i \mid (i,j) \in R_1\}$ and $\mathcal{C}_j = \{k \mid (j,k) \in R_2\}$. By definition, $Z = \bigcup_j \mathcal{A}_j \times \mathcal{C}_j$. The size of Z depends crucially on the extent of overlap among the sets $\{\mathcal{A}_j \times \mathcal{C}_j\}_j$. However, the total size of these sets may be much larger than both input and output (see [8]), so any approach that explicitly processes them is unattractive.

The starting point for our improved estimation algorithm is a well-known algorithm for estimating the number of distinct elements in a data streaming context [10]. (We remark that the idea underlying this algorithm is similar to that of Cohen [22].) Our main insight is that this algorithm can be extended such that a set of the form $\mathcal{A}_j \times \mathcal{C}_j$ can be added to the sketch in expected time $\mathcal{O}(|\mathcal{A}_j| + |\mathcal{C}_j|)$, i.e., without explicitly generating all pairs. The idea is to use a hash function that is particularly well suited for the purpose: sufficiently structured to make hash values easy to handle algorithmically, and sufficiently random to make the analysis of sketching accuracy go through.

3.1.1 Motivation

Cohen [23] investigated the use of the size estimation technique in sparse matrix computations. In particular, it can be used to find the optimal order of multiplying sparse matrices, and in memory allocation for sparse matrix computations.

In addition, we are motivated by applications in database systems, where size estimation is an important part of query optimization. Examples of database queries that correspond to Boolean matrix products are:

- A query that computes all pairs of people in a social network with a distance 2 connection ("possible friends").
- A query to compute all director-actor pairs who have done at least one movie together.

- In a business database with information on orders, and a categorization of products into types, compute the relation that contains a tuple (c, p) if customer c has made an order for a product of type p.

As a final example, we consider a fundamental data mining task. Given a list of sets, the famous Apriori data mining algorithm [4] finds frequent item pairs by counting the number occurrences of item pairs where each single element is frequent. So if $R_1 = R_2$ denotes the relationship between high-support (i.e., frequent) items and sets in which they occur, Z is exactly the pairs of frequent items, and the number of distinct items in Z determines the space usage of Apriori. Since Apriori may be very time consuming, it is of interest to establish whether sufficient space is available before choosing the support threshold and running the algorithm.

3.1.2 Further related work

JD sketch.

Ganguly et al. [38] previously considered techniques that compute a data structure (a *sketch*) for R_1 and R_2 (individually), such that the two sketches suffice to compute an approximation of z.

Define $n_a = |\{i \mid \exists j : (i, j) \in R_1\}|$ and $n_c = |\{k \mid \exists j : (j, k) \in R_2\}|$. Ganguly et al. show that for any constant c and any β, a sketching method that returns a c-approximation with probability $\Omega(1)$ whenever $z \geq \beta$ must, on a worst-case input, use expected space

$$\Omega(\min(n_1 + n_2, n_a n_c(n_1/n_a + n_2/n_c)/\beta)) \\ = \Omega(\min(n_1 + n_2, (n_1 n_c + n_2 n_a)/\beta)) \text{ bits.}$$

The lower bound proof applies to the case where $n_1 = n_2$, $n_a = n_c$, and $z < n_a + n_c$. We note that [38] claims a stronger lower bound, but their proof does not establish a lower bound above $n_1 + n_2$ bits. Ganguly et al. present a sketch whose worst-case space usage matches the lower bound times polylogarithmic factors (while not stated in [38], the trivial sketch that stores the whole input can be used to nearly match the first term in the minimum).

In Section 3.3 we analyze a simple sketch, previously considered in other contexts by Gibbons [40] and Ganguly and Saha [37]. It similarly matches the above worst-case bound, but the exact space usage is incomparable to that of [38].

The focus of [38] is on space usage, and so the time for updating sketches, and for computing the estimate from two sketches, is not discussed in the paper. Looking at the data structure description we see that the update time grows linearly with the quantity they call s_1, which is $\Omega(n)$ in the worst case. Also, the sketch uses a number of summary data structures

that are accessed in a random fashion, meaning that the worst case number of I/Os is at least $\Omega(n)$ *unless* the sketch fits internal memory. By the above lower bound we see that keeping the sketch in internal memory is not feasible in general. In contrast, the sketch we consider allows collection and combination of sketches to be done efficiently in linear time and I/O.

Distinct elements and distinct paths estimation.

Our work is related in terms of techniques to papers on estimating the number of distinct items in a data stream (see [10] and its references). However, our basic estimation algorithm does not work in a general streaming model, since it crucially needs the ability to access all tuples with a particular value on the join attribute together.

Ganguly and Saha [37] consider the problem of estimating the number of distinct vertex pairs connected by a length-2 path in a graph whose edges are given as a data stream of n edges. This corresponds to size estimation for the special case of *squaring* a matrix (or self-join in database terminology). It is shown that space $\sqrt{n}$ is required, and that space roughly $\mathcal{O}(n^{3/4})$ suffices for constant ε (unless there are close to n connected components). The estimation itself is a join-distinct size estimation of a sample of the input having size no smaller than $\mathcal{O}(n^{3/4}/\varepsilon^2)$. Using Cohen's estimation algorithm this would require time $\mathcal{O}(n^{3/4}/\varepsilon^4)$, so this is $\mathcal{O}(n)$ time only for $\varepsilon > 1/\sqrt[16]{n}$.

Join synopses.

Acharya et al. [1] proposed so-called *join synopses* that provide a uniform sample of the result of a join. While this can be used to estimate result sizes of a variety of operations, it does not seem to yield efficient estimates of join-project sizes. The reason is that a standard uniform sample is known to be inefficient for estimating the number of distinct values [19]. In addition, Acharya et al. assume the presence of a foreign-key relationship, i.e., that each tuple has at most one matching tuple in the other table(s), which is also known as a *snow flake* schema. Our method has no such restriction.

Distinct sampling.

Gibbons [40] considered different samples that can be extracted by a scan over the input, and proposed *distinct samples*, which offer much better guarantees with respect to estimating the number of distinct values in query results. Gibbons shows that this technique applies to single relations, and to foreign key joins where the join result has the same number of tuples as one of the relations. In Section 3.3 we show that the distinct samples, with suitable settings of parameters, can often be used in our setting to get an accurate estimate of $z = |Z|$. The processing of two distinct samples to

produce the estimate consists of running the efficient estimation algorithm of Section 3.2 on the samples, meaning that this is time- and I/O-efficient.

3.2 Our algorithm

The task is to estimate the size z of $Z = \pi_{ac}(R_1 \bowtie R_2)$. We may assume that attribute values are $\mathcal{O}(\log n)$-bits integers, since any domain can be mapped into this one using hashing, without changing the join result size with high probability. When discussing I/O bounds, B is the number of such integers that fits in a disk block. In linear expected time (by hashing) or sort(n) I/Os we can cluster the relations according to the value of the join attribute b. By initially eliminating input tuples that do not have any matching tuples in the other relation we may assume without loss of generality that $z \geq n/2$.

In what follows, k is a positive integer parameter that determines the space usage and accuracy of our method. The technique used is to compute the kth smallest value v of a hash function $h(x, y)$, for $(x, y) \in Z$. Analogously to the result by Bar-Yossef et al. [10] we can then use $\tilde{z} = k/v$ as an estimator for z.

Our main building block is an efficient iteration over all tuples $(x, \cdot, y) \in R_1 \bowtie R_2$ for which $h(x, y)$ is smaller than a carefully chosen threshold p, and is therefore a candidate for being among the k smallest hash values. The essence of our result lies in how the pairs being output by this iteration are computed in expected linear time. We also introduce a new buffering trick to update the sketch in expected amortized $\mathcal{O}(1)$ time per pair. In a nutshell, each time k new elements have been retrieved, they are merged using a linear time selection procedure with the previous k smallest values to produce a new (unordered) list of the k smallest values.

Theorem 3.8 *Let $R_1(a, b)$ and $R_2(b, c)$ be relations with n tuples in total, and define $z = |\pi_{ac}(R_1 \bowtie R_2)|$. Let ε, $0 < \varepsilon < \frac{1}{2}$ be given. There are algorithms that run in expected $\mathcal{O}(n)$ time on a RAM, and expected $\mathcal{O}(sort(n))$ I/Os in the cache-oblivious model, and output a number $\tilde{z}$ such that for $k = 9/\varepsilon^2$:*

- $\mathbf{Pr}[(1 - \varepsilon)z < \tilde{z} < (1 + \varepsilon)z] \geq 2/3$ *when* $z > k^2$, *and*
- $\mathbf{Pr}[\tilde{z} < (1 + \varepsilon)k^2] \geq 2/3$ *when* $z \leq k^2$.

Observe that for $\varepsilon > 4/\sqrt[4]{n}$, since $z \geq n/2$ we will be in the first case, and get the desired $1 \pm \varepsilon$ approximation with probability 2/3. The error probability can be reduced from 1/3 to δ by the standard technique of doing $\mathcal{O}(\log(1/\delta))$ runs and taking the median (the analysis follows from a Chernoff bound). We remark that this can be done in such a way that the $\mathcal{O}(\log(1/\delta))$ factor affects only the RAM running time and not the number of I/Os. For constant relative error $\varepsilon > 0$ we have the following result:

Theorem 3.9 *In the setting of Theorem 3.8, if ε is constant there are algorithms that run in expected $\mathcal{O}(n)$ time on a RAM, and expected $\mathcal{O}(sort(n))$ I/Os in the cache-oblivious model, that output $\tilde{z}$ such that* $\mathbf{Pr}[(1-\varepsilon)z < \tilde{z} < (1+\varepsilon)z] = \mathcal{O}(1/\sqrt{n})$.

The error probability can be reduced to n^{-c} for any desired constant c by running the algorithms $\mathcal{O}(c)$ times, and taking the median as above.

3.2.1 Finding pairs

For $\mathcal{B} = \pi_b(R_1) \cup \pi_b(R_2)$ and each $i \in \mathcal{B}$ let $\mathcal{A}_i = \pi_a(\sigma_{b=i}(R_1))$ and $\mathcal{C}_i = \pi_c(\sigma_{b=i}(R_2))$. We would like to efficiently iterate over all pairs $(x, y) \in \mathcal{A}_i \times \mathcal{C}_i$, $i \in \mathcal{B}$, for which $h(x, y)$ is smaller than a threshold p. This is done as follows (see Algorithm 1 for pseudocode).

For a set U, let $h_1, h_2 : U \to [0; 1]$ be hash functions chosen independently at random from a pairwise independent family, and define $h : U \times U \to [0; 1]$ by[2]

$$h(x, y) = (h_1(x) - h_2(y)) \bmod 1.$$

According to Lemma 3.6 on page 8, h is also a pairwise independent hash function — a property we will utilize later. Now, conceptually arrange the values of $h(x, y)$ in an $|\mathcal{A}_i| \times |\mathcal{C}_i|$ matrix, and order the rows by increasing values of $h_1(x)$, and the columns by increasing values of $h_2(y)$. Then the values of $h(x, y)$ will decrease (modulo 1) from left to right, and increase (modulo 1) from top to bottom.

For each $i \in \mathcal{B}$, we traverse the corresponding $|\mathcal{A}_i| \times |\mathcal{C}_i|$ matrix by visiting the columns from left to right, and in each column t finding the row $\bar{s}$ with the smallest value of $h(x_{\bar{s}}, y_t)$. Values smaller than p in that column will be found in rows subsequent to $\bar{s}$. When all such values have been output, the search proceeds in column $t + 1$. Notice, that if $h(x_{\bar{s}}, y_t)$ was the minimum value in column t, then the minimum value in column $t + 1$ is found by increasing $\bar{s}$ until $h(x_{\bar{s}}, y_{t+1}) < h(x_{(\bar{s}-1) \bmod |\mathcal{A}_i|}, y_{t+1})$. We observe that the algorithm is robust to decreasing the value of the threshold p during execution, in the sense that the algorithm still outputs all pairs with hash value at most p.

3.2.2 Estimating the size

While finding the relevant pairs, we will use a technique that allows us to maintain the k smallest hash values in an unordered buffer instead of using a heap data structure (lines 14–18 in Algorithm 1). In this way we are able to maintain the k smallest hash values in constant amortized time per

[2] We observe that this is different from the "composable hash functions" used by Ganguly et al. [38].

Algorithm 1 Pseudocode for the size estimator.

```
1: procedure DisItems(p, ε)
2:     k ← ⌈9/ε²⌉
3:     F ← ∅
4:     for i ∈ B do
5:         x ← A_i sorted according to h1-value
6:         y ← C_i sorted according to h2-value
7:         s̄ ← 1
8:         for t := 1 to |C_i| do
9:             while h(x_s̄, y_t) > h(x_{(s̄−1) mod |A_i|}, y_t) do      ▷ Find s̄ s.t. h(x_s̄, y_t) is min.
10:                s̄ ← (s̄ + 1) mod |A_i|
11:            end while
12:            s ← s̄
13:            while h(x_s, y_t) < p do                              ▷ Find all s where h(x_s, y_t) < p
14:                F ← F ∪ {(x_s, y_t)}
15:                if |F| = k then          ▷ Buffer filled, find smallest hash values in S ∪ F
16:                    (p, S) ← Combine(S, F)
17:                    F ← ∅
18:                end if
19:                s ← (s + 1) mod |A_i|
20:            end while
21:        end for
22:    end for
23:    (p, S) ← Combine(S, F)
24:    if |S| = k then
25:        return "z̃ = k/p and z̃ ∈ [(1 ± ε)z] with probability 2/3"
26:    else
27:        return "z̃ = k², z ≤ k² with probability 2/3"
28:    end if
29: end procedure

30: procedure Combine(S, F)
31:    v ← Rank(h(S) ∪ h(F), k)                  ▷ Rank(·, k) returns the kth smallest value
32:    S ← {x ∈ S ∪ F | h(x) ≤ v}
33:    return (v, S)
34: end procedure
```

insertion in the buffer, eliminating the $\log k$ factor implied by the heap data structure.

Let S and F be two unordered sets containing, respectively, the k smallest hash values seen so far (all, of course, smaller than p), and the latest up to k elements seen. We avoid duplicates in S and F (i.e., the sets are kept disjoint) by using a simple hash table to check for membership before insertion. Whenever $|F| = k$ the two sets S and F are combined in order to obtain a new sketch S. This is done by finding the median of $S \cup F$, which takes $\mathcal{O}(k)$ time using either deterministic methods (see [31]) or more practical randomized ones [47].

At each iteration the current kth smallest value in S may be smaller than the initial value p, and we use this as a better substitute for the initial value of p. However, in the analysis below we will upper bound both the

running time and the error probability using the initial threshold value p.

3.2.3 Time analysis

We split the time analysis into two parts. One part accounts for iterations of the inner while loop in lines 13–20, and the other part accounts for everything else. We first consider the RAM model, and then outline the analysis in the cache-oblivious model.

Inner while loop. Observe that for each iteration, one pair (x_s, y_t) is added to F (if it is not already there). For each $t \in \mathcal{C}_i$, $p|\mathcal{A}_i|$ elements are expected to be added since each pair (x_s, y_t) is added with probability p. This means that the expected total number of iterations is $\mathcal{O}(p|\mathcal{A}_i||\mathcal{C}_i|)$. Each call to COMBINE costs time $\mathcal{O}(k)$, but we notice that there must be at least k iterations between successive calls, since the size of F must go from 0 to k. Inserting a new value into F costs $\mathcal{O}(1)$ since the set is not sorted. Hence, the total cost of the inner loop is $\mathcal{O}(p|\mathcal{A}_i||\mathcal{C}_i|)$.

Remaining cost. Consider the processing of a single $i \in \mathcal{B}$ in Algorithm 1. The initial sorting of hash values can be done with bucket sort requiring expected time $\mathcal{O}(|\mathcal{A}_i|+|\mathcal{C}_i|)$ since the numbers sorted are pairwise independent (by the same analysis as for hashing with chaining).

For the iteration in lines 9–11 observe that $h(x_{\bar{s}}, y_t)$ is monotone modulo 1, and we have at most a total of $2|\mathcal{A}_i|$ increments of $\bar{s}$ among all $t \in \mathcal{C}_i$. Thus, the total number of iterations is $\mathcal{O}(|\mathcal{A}_i|)$, and the total cost for each $i \in \mathcal{B}$ is $\mathcal{O}(|\mathcal{A}_i| + |\mathcal{C}_i|)$.

The time for the final call to COMBINE is dominated by the preceding cost of constructing S and F.

I/O efficient variant. As for I/O efficiency, notice that a direct implementation of Algorithm 1 may cause a linear number of cache misses if $\mathcal{A}_i$ and $\mathcal{C}_i$ do not fit into internal memory. To get an I/O-efficient variant we use a cache-oblivious sorting algorithm, sorting R_1 according to $(b, h_1(a))$, and R_2 according to $(b, h_2(c))$, such that the sorting steps for each $i \in \mathcal{B}$ is replaced by one global sorting step.

The rest of the algorithm works directly in a cache-oblivious setting. To see this, notice that it suffices to keep in internal memory the two input blocks that are closest to each of the pointers s, t, and $\bar{s}$. The cache-oblivious model assumes the cache to behave in an optimal fashion, so also in this model there will be $\Omega(B)$ operations between cache misses, and $\mathcal{O}(n/B)$ I/Os, expected, in total.

Lemma 3.10 *Suppose $R_1(a, b)$ and $R_2(b, c)$ are relations with n tuples in total. Let $p > 0$ and $\varepsilon > 0$ be given. Then Algorithm 1 runs in expected*

$\mathcal{O}(n+\sum_i p|\mathcal{A}_i||\mathcal{C}_i|)$ time and $\mathcal{O}(1/\varepsilon^2)$ space on a RAM, and can be modified to use expected $\mathcal{O}(sort(n))$ I/Os in the cache-oblivious model.

Choice of threshold p.

We would like a value of p that ensures the expected processing time is $\mathcal{O}(n)$. At the same time p should be large enough that we expect to reach line 25 where an exact estimate is returned (except possibly in the case where z is small).

Lemma 3.11 *Let $j \in \mathcal{B}$ satisfy $|\mathcal{A}_i||\mathcal{C}_i| \leq |\mathcal{A}_j||\mathcal{C}_j|$ for all $i \in \mathcal{B}$. Then $p = \min(1/k, k/(|\mathcal{A}_j||\mathcal{C}_j|))$ gives an expected $\mathcal{O}(n)$ running time for Algorithm 1.*

Proof We argue that for each i, $p|\mathcal{A}_i||\mathcal{C}_i| \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$, which by Lemma 3.10 implies running time

$$\mathcal{O}(n + \sum_i p|\mathcal{A}_i||\mathcal{C}_i|) = \mathcal{O}(n + \sum_i \max(|\mathcal{A}_i|, |\mathcal{C}_i|)) = \mathcal{O}(n).$$

Suppose first that $|\mathcal{A}_i||\mathcal{C}_i| \geq k^2$. Then $p = k/(|\mathcal{A}_j||\mathcal{C}_j|)$ and $p|\mathcal{A}_i||\mathcal{C}_i| \leq k \leq \sqrt{|\mathcal{A}_i||\mathcal{C}_i|} \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$. Otherwise, when $|\mathcal{A}_i||\mathcal{C}_i| < k^2$, we have $p = 1/k$ and $p|\mathcal{A}_i||\mathcal{C}_i| = |\mathcal{A}_i||\mathcal{C}_i|/k \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$. □

We note that when R_1 and R_2 are sorted according to b, the value of p specified above can be found by a simple scan over both inputs. Our experiments indicate that in practice this initial scan is not needed, see Section 3.4 for details.

3.2.4 Error probability

Theorem 3.12 *Let h be a pairwise independent hash function. Suppose we are provided with a stream of elements N with $h(x) < v$ for all $x \in N$. Further, let ε, $0 < \varepsilon < \frac{1}{2}$ be given and assume that $p \geq \min\left(\frac{k}{2z}, \frac{1}{k}\right)$, where $k \geq 9/\varepsilon^2$, and z is the number of distinct items in N. Then Algorithm 1 produces an approximation $\tilde{z}$ of z such that*

- $\mathbf{Pr}[(1-\varepsilon)z < \tilde{z} < (1+\varepsilon)z] \geq 2/3$ *for $z > k^2$, and*
- $\mathbf{Pr}[\tilde{z} < (1+\varepsilon)k^2] \geq 2/3$ *for $z \leq k^2$.*

Proof The error probability proof is similar to the one that can be found in [10], with some differences and extensions. We bound the error probability of three cases: the estimate being smaller/larger than the multiplicative error bound, and the number of obtained samples being too small.

Estimate too large. Let us first consider the case where $\tilde{z} > (1+\varepsilon)z$, i.e. the algorithm overestimates the number of distinct elements. This happens if the stream N contains at least k entries smaller than $k/(1+\varepsilon)z$. For each pair $(a,c) \in Z$ define an indicator random variable $X_{(a,c)}$ as

$$X_{(a,c)} = \begin{cases} 1 & h(a,c) < k/(1+\varepsilon)z \\ 0 & \text{otherwise} \end{cases}$$

That is, we have z such random variables for which the probability of $X_{(a,c)} = 1$ is exactly $k/(1+\varepsilon)z$ and $\mathbf{E}[X_{(a,c)}] = k/(1+\varepsilon)z$. Now define $Y = \sum_{(a,c)\in Z} X_{(a,c)}$ so that $\mathbf{E}[Y] = \mathbf{E}[\sum_{(a,c)\in Z} X_{(a,c)}] = \sum_{(a,c)\in Z} \mathbf{E}[X_{(a,c)}] = k/(1+\varepsilon)$. By the pairwise independence of the $X_{(a,c)}$ we also get $\mathbf{Var}(Y) \leq k/(1+\varepsilon)$ due to Lemma 3.7. Using Chebyshev's inequality [65] we can bound the probability of having too many pairs reported:

$$\mathbf{Pr}\left[Y > k\right] \leq \mathbf{Pr}\left[|Y - \mathbf{E}[Y]| > k - \tfrac{k}{1+\varepsilon}\right] \leq \frac{\mathbf{Var}[Y]}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \frac{k/(1+\varepsilon)}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \tfrac{1}{6}$$

since $k \geq 9/\varepsilon^2$.

Estimate too small. Now, consider the case where $\tilde{z} < (1-\varepsilon)z$ which happens when at most k hash values are smaller than $k/(1-\varepsilon)z$ and at least k hash values are smaller than p. Define $X'_{(a,c)}$ as

$$X'_{(a,c)} = \begin{cases} 1 & h(a,c) < k/(1-\varepsilon)z \\ 0 & \text{otherwise} \end{cases}$$

so that $\mathbf{E}[X'_{(a,c)}] = k/(1-\varepsilon)z < (1+\varepsilon)k/z$. Moreover, with $Y' = \sum_{(a,c)\in Z} X'_{(a,c)}$ we have $\mathbf{E}[Y'] = k/(1-\varepsilon)$, and since the indicator random variables defined above are pairwise independent, we also have $\mathbf{Var}[Y'] \leq \mathbf{E}[Y'] < (1+\varepsilon)k$. Chebyshev's inequality gives:

$$\begin{aligned} \mathbf{Pr}\left[Y' > k\right] &\leq \mathbf{Pr}\left[|Y' - \mathbf{E}[Y']| > \tfrac{k}{1-\varepsilon} - k\right] \\ &\leq \frac{\mathbf{Var}[Y']}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \frac{(1+\varepsilon)k}{\left(\frac{k}{1-\varepsilon} - k\right)^2} < \tfrac{1}{9} \end{aligned}$$

since $k \geq 9/\varepsilon^2$.

Not enough samples. Consider the case where $|S| < k$ after all pairs have been retrieved. In this case the algorithm returns $\beta = k^2$ as an upper bound on the number of distinct elements in the output, and we have two possible situations: either there is actually less than k^2 distinct pairs in the output, in which case the algorithm is correct, or there are more than k^2 distinct elements in the output, in which case it is incorrect. In the latter

case, less than k hash values have been smaller than p and the kth smallest value v is therefore larger than p. Define $X''_{(a,c)}$ as

$$X''_{(a,c)} = \begin{cases} 1 & h(a,c) < p \\ 0 & \text{otherwise} \end{cases}$$

and let again $Y'' = \sum_{(a,c)\in Z} X''_{(a,c)}$. It results that $\mathbf{E}[X''_{(a,c)}] = p$ and $\mathbf{E}[Y''] = zp$, and because of pairwise independancy of $X''_{(a,c)}$, also $\mathbf{Var}[Y''] \leq \mathbf{E}[Y'']$. Using Chebyshev's inequality and remembering that $z > k^2$ in this case we have:

$$\begin{aligned} \mathbf{Pr}[Y'' < k] &\leq \mathbf{Pr}[|Y'' - \mathbf{E}[Y'']| > zp - k] \\ &\leq \frac{zp}{(zp-k)^2} \leq \frac{zp}{\left(\frac{1}{2}zp\right)^2} \leq 2/k \leq 1/18. \end{aligned}$$

using that $k \geq 9/\varepsilon^2 \geq 36$.

In conclusion, the probability that the algorithm fails to output an estimate within the given limits is at most $1/6 + 1/9 + 1/18 = 1/3$. □

For the proof of Theorem 3.9 we observe that in the above proof, if ε is constant the error probability is $\mathcal{O}(1/k)$. Using $k = \sqrt{n}$ we get linear running time and error probability $\mathcal{O}(1/\sqrt{n})$.

Realization of hash functions.

We have used the idealized assumption that hash values were real numbers in $(0;1)$. Let $m = n^3$. To get an actual implementation we approximate (by rounding down) the real numbers used by rational numbers of the form i/m, for integer i. This changes each hash value by at most $2/m$. Now, because of the way hash values are computed, the probability that we get a different result when comparing two real-valued hash values and two rational ones is bounded by $2/m$. Similarly, the probability that we get a different result when looking up a hash value in the dictionary is bounded by $2k/m$. Thus, the probability that the algorithm makes a different decision based on the approximation, in any of its steps, is $\mathcal{O}(kn/m) = o(1)$. Also, for the final output the error introduced by rounding is negligible.

3.3 Distinct sketches

A well-known approach to size estimation in, described in generality by Gibbons [40] and explicitly for join-project operations in [8, 37], is to sample random subsets $R'_1 \subseteq R_1$ and $R'_2 \subseteq R_2$, compute $Z' = \pi_{ac}(R'_1 \bowtie R'_2)$, and use the size of Z' to derive an estimate for z. This is possible if $R'_1 = \sigma_{a\in S_a}(R_1)$, where $S_a \subseteq \pi_a(R_1)$ is a random subset where each element is

picked independently with probability p_1, and similarly $R_2' = \sigma_{c \in S_c}(R_2)$, where $S_c \subseteq \pi_c(R_2)$ includes each element independently with probability p_2. Then $z' = |Z'|/(p_1 p_2)$ is an unbiased estimator for z. The samples can be obtained in small space using hash functions whose values determine which elements are picked for S_a and S_c. The value $|Z'|$ can be approximated in linear time using the method described in section 3.2 if the samples are sorted — otherwise one has to add the cost of sorting. In either case, the estimation algorithm is I/O-efficient.

Below we analyze the variance of the estimator z', to identify the minimum sampling probability that introduces only a small relative error with good probability. The usual technique of repetition can be used to reduce the error probability. Recall that we have two relations with n_1 and n_2 tuples, respectively, and that n_a and n_c denotes the number of distinct values of attributes a and c, respectively. Our method will pick samples R_1' and R_2' of expected size s from each relation, where $s = p_1 n_1 = p_2 n_2$ is a parameter to be specified.

Theorem 3.13 *Let R_1' and R_2' be samples of size s, obtained as described above. Then $z' = |\pi_{ac}(R_1' \bowtie R_2')|/(p_1 p_2)$ is a $1 \pm \varepsilon$ approximation of $z = |\pi_{ac}(R_1 \bowtie R_2)|$ with probability $5/6$ if $z > \beta$, where $\beta = \frac{14}{\varepsilon^2}\left(\frac{n_c n_1 + n_a n_2}{s}\right)$. If $z \leq \beta$ then $z' < (1+\varepsilon)\beta$ with probability $5/6$.*

3.3.1 Analysis of variance

To arrive at a sufficient condition that z' is a $1 \pm \varepsilon$ approximation of z with good probability, we analyze its variance. To this end define $Z_{i\cdot} = \{j \,|\, (i,j) \in Z\}$, $Z_{\cdot j} = \{i \,|\, (i,j) \in Z\}$, and let

$$X_i = \begin{cases} 1 - p_1, & \text{if } i \in S_a \\ -p_1, & \text{otherwise} \end{cases} \qquad Y_j = \begin{cases} 1 - p_2, & \text{if } j \in S_c \\ -p_2, & \text{otherwise} \end{cases}.$$

By definition of S_a, $\mathbf{E}[X_i] = \mathbf{Pr}[i \in S_a](1-p_1) - \mathbf{Pr}[i \notin S_a]p_1 = 0$. Similarly, $\mathbf{E}[Y_i] = 0$. We have that $(i,j) \in Z'$ if and only if $(i,j) \in Z$ and $(i,j) \in S_a \times S_c$. This means that $z' p_1 p_2 = \sum_{(i,j) \in Z}(X_i + p_1)(Y_j + p_2)$. By linearity of expectation, $\mathbf{E}[(X_i + p_1)(Y_j + p_2)] = p_1 p_2$, and we can write the variance of $z' p_1 p_2$, $\mathbf{Var}(z' p_1 p_2)$ as

$$\mathbf{E}\left[\left(\sum_{(i,j) \in Z} ((X_i + p_1)(Y_j + p_2) - p_1 p_2)\right)^2\right].$$

Expanding the product and using linearity of expectation, we get

$$\begin{aligned}\mathbf{Var}(z'p_1p_2) = &\sum_{(i,j)\in Z}\sum_{(i,j')\in Z}\mathbf{E}\left[X_i^2p_2^2\right] + \sum_{(i,j)\in Z}\sum_{(i',j)\in Z}\mathbf{E}\left[Y_j^2p_1^2\right]\\ &+\sum_{(i,j)\in Z}\mathbf{E}\left[X_i^2Y_j^2\right]\\ =&\sum_{i\in\mathcal{A}}\sum_{j,j'\in Z_{i\cdot}}p_2^2\,\mathbf{E}\left[X_i^2\right]+\sum_{j\in\mathcal{C}}\sum_{i,i'\in Z_{\cdot j}}p_1^2\,\mathbf{E}\left[Y_i^2\right]\\ &+z\,\mathbf{E}\left[X_i^2\right]\mathbf{E}\left[Y_i^2\right].\end{aligned}$$

Since $\mathbf{E}\left[X_i^2\right] = p_1(1-p_1)^2 + (1-p_1)(-p_1)^2 = p_1 - p_1^2 < p_1$, and similarly $\mathbf{E}\left[Y_j^2\right] < p_2$ we can upper bound $\mathbf{Var}(z')$ as follows:

$$\begin{aligned}\mathbf{Var}(z') &= (p_1p_2)^{-2}\,\mathbf{Var}(z'p_1p_2)\\ &< (p_1p_2)^{-2}\Big(\sum_{i\in\mathcal{A}}\sum_{j,j'\in Z_{i\cdot}}p_1p_2^2+\sum_{j\in\mathcal{C}}\sum_{i,i'\in Z_{\cdot j}}p_1^2p_2+z\,p_1p_2\Big)\\ &\le (p_1p_2)^{-2}\left(n_cz\,p_1p_2^2+n_az\,p_1^2p_2+z\,p_1p_2\right)\\ &= \left(n_c/p_1+n_a/p_2+(p_1p_2)^{-1}\right)\,z\ .\end{aligned}$$

3.3.2 Sufficient sample size

We are ready to derive a bound on the probability that z' deviates significantly from z. Choose $0 < \varepsilon < 1$. Since $z = \mathbf{E}[z']$ Chebyshev's inequality says

$$\mathbf{Pr}[|z'-z] > \varepsilon z] < \frac{\mathbf{Var}(z')}{(\varepsilon z)^2} \le \left(n_c/p_1+n_a/p_2+(p_1p_2)^{-1}\right)/(\varepsilon^2 z).$$

This can equivalently be expressed in terms of the sample size s, since $p_1 = s/n_1$ and $p_2 = s/n_2$:

$$\mathbf{Pr}[|z'-z] > \varepsilon z] < \left(n_cn_1+n_an_2+n_1n_2/s\right)/(s\varepsilon^2 z).$$

We seek a sufficient condition on s that the above probability is bounded by some constant $\delta < \frac{1}{2}$ (e.g. $\delta = 1/6$). In particular it must be the case that $n_1n_2/(s^2\varepsilon^2 z) < \delta$, which implies $s > \sqrt{n_1, n_2/(\delta z)} \ge \sqrt{n_1, n_2/(\delta n_a n_c)}$. Hence, using the arithmetic-geometric inequality:

$$n_1n_2/s < \sqrt{n_cn_1n_an_2\delta} \le (n_cn_1+n_an_2)/(2\sqrt{\delta}).$$

In other words, it suffices that

$$\begin{aligned}&\frac{\left(n_cn_1+n_an_2\right)\left(1+(2\sqrt{\delta})^{-1}\right)}{s\varepsilon^2 z} < \delta\\ &\qquad\qquad\iff s > \left(\frac{n_cn_1+n_an_2}{z}\right)\left(\frac{1+(2\sqrt{\delta})^{-1}}{\varepsilon^2\delta}\right).\end{aligned}$$

One apparent problem is the chicken-egg situation: z is not known in advance. If a lower bound on z is known, this can be used to compute a sufficient sample size. Alternatively, if we allow a larger relative error whenever $z \leq \beta$ we may compute a sufficient value of s based on the assumption $z \geq \beta$. Whenever $z < \beta$ we then get the guarantee that $z' < (1+\varepsilon)\beta$ with probability $1-\delta$. Theorem 3.13 follows by fixing s and solving for β.

Optimality.

For constant ε and δ our upper bound matches the lower bound of Ganguly et al. [38] whenever this does not exceed $n_1 + n_2$. It is trivial to achieve a sketch of size $\mathcal{O}((n_1+n_2)\log(n_1+n_2))$ bits (simply store hash signatures for the entire relations). We also note that the lower bound proof in [38] uses certain restrictions of parameters ($n_1 = n_2$, $n_a = n_c$, and $z < n_a + n_c$), so it may be possible to do better in some settings.

3.4 Experiments

We have run our algorithm on most of the datasets from the Frequent Itemset Mining Implementations (FIMI) Repository[3] together with some datasets extracted from the Internet Movie Database (IMDB). Each dataset represents a single relation, and motivated by the Apriori space estimation example in the introduction, we perform the size estimation on self-joins of these relations. Table 3.1 displays the size of each dataset together with the number of distinct a- and c-values.

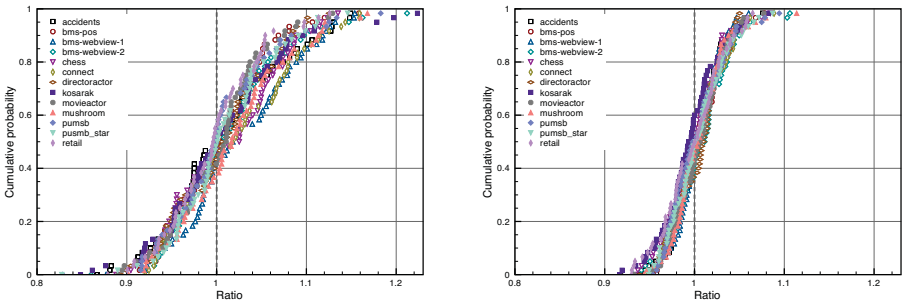
Rather than selecting h_1 and h_2 from an arbitrary pairwise independent family, we store functions that map the attribute values to fully random and independent values of the form $d/2^{64}$, where d is a 64 bit random integer formed by reading 64 random bits from the Marsaglia Random Number CDROM[4].

We have chosen an initial value of $p = 1$ for our tests in order to be certain to always arrive at an estimate. In most cases we observed that p quickly decreases to a value below $1/k$ anyway. But as the sampling probability decreases, the probability that the sketch will never be filled increases, implying that we will not get a linear time complexity with an initial value of $p = 1$. In the cases where the sketch is not filled, we report $|F|/(p_1 p_2)$ as the estimate, where $|F|$ is the number of elements in the buffer.

Tests have been performed for $k = 256$ and $k = 1024$. In each test, 60 independent estimates were made and compared to the exact size of the join-project. By sorting the ratios "estimate"/"exact size" we can draw the cumulative distribution function for each instance that, for each ratio-value

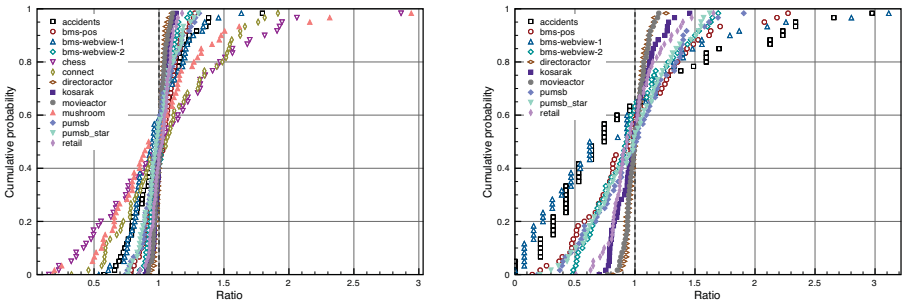
[3] http://fimi.cs.helsinki.fi

[4] http://www.stat.fsu.edu/pub/diehard/



(a) $k = 256$ (b) $k = 1024$

Figure 3.1: The cumulative distribution functions for $k = 256$ and $k = 1024$. It is seen that $k = 1024$ yields a more precise estimate than $k = 256$ with 2/3 of the estimates being within 4% and 10% of the exact size, respectively.



(a) $k = 1024$, $p_1 = p_2 = 0.1$ (b) $k = 1024$, $p_1 = p_2 = 0.01$

Figure 3.2: Plots for sampling with probability 10% and 1%. If the sampling probability is too small, no elements at all may reach the sketch and in these cases we are not able to return an estimate. Instances with no estimates have been left out of the graph.

Instance	z	$n_a\ (= n_c)$	$\varepsilon_{0.1}$	$\varepsilon_{0.01}$
accidents	$94 \cdot 10^3$	468	1.18	3.73
bms-pos	$760 \cdot 10^3$	1,657	0.78	2.47
bms-webview-1	$128 \cdot 10^3$	497	1.04	3.29
bms-webview-2	$1.45 \cdot 10^6$	3,340	0.80	2.54
chess	$5.24 \cdot 10^3$	75	2.00	6.33
connect	$13.8 \cdot 10^3$	129	1.62	5.12
directoractor	$734 \cdot 10^6$	50,645	0.14	0.44
kosarak	$66.2 \cdot 10^6$	41,270	0.42	1.32
movieactor	$111 \cdot 10^6$	51,226	0.36	1.14
mushroom	$7.17 \cdot 10^3$	119	2.16	6.82
pumsb	$1.07 \cdot 10^6$	2,113	0.74	2.35
pumsb_star	$967 \cdot 10^3$	2,088	0.78	2.46
retail	$7.19 \cdot 10^6$	16,470	0.80	2.53

Table 3.1: Characteristics of the used datasets. The rightmost middle column displays the size $n_a = |\pi_a(R_1)|$ (which in this case is equals $n_c = |\cup\, \pi_c(R_2)|$). The two rightmost columns display the theoretical error as described in Theorem 3.13, for $p_1 = p_2 = 0.1$ and $p_1 = p_2 = 0.01$, respectively. These theoretical error bounds, which hold with probability 5/6, are significantly larger than the actual observed errors in Figure 3.2.

on the x-axis, displays on the y-axis the probability that an estimate will have this ratio or less. Figure 3.1 shows plots for $k = 256$ and $k = 1024$. In Table 3.2 we compare the theoretical error ε with observed error for 2/3 of the results. As seen, the observed error is smaller than the theoretical upper bound.

In Figure 3.2 we perform sampling with 10% and 1% probability, as described in Section 3.3. Again, the samples are chosen using truly random bits. The variance of estimates increase as the probability decreases, but increases more for smaller than for larger instances. If the sampling probability is too small, no elements at all may reach the sketch and in these cases we are not able to return an estimate. As seen, the observed errors in the figure are significantly smaller than the theoretical errors seen in Table 3.1.

k	ϵ	Observed ε
256	0.188	0.1
1024	0.094	0.04

Table 3.2: The theoretical error bound is $\varepsilon = \sqrt{9/k}$ as stated Theorem 3.12. The observed error in Figure 3.1, however, is significantly less.

3.5 Conclusion

We have presented improved algorithms for estimating the size of Boolean matrix products, for the first time allowing $o(1)$ relative error to be achieved in linear time. An interesting open problem is if this can be extended to transitive closure in general graphs, and/or to products of more than two matrices.

Acknowledgements. We would like to thank Jelani Nelson for useful discussions, and in particular for introducing us to the idea of buffering to achieve faster data stream algorithms. Also, we thank Sumit Ganguly for clarifying the lower bound proof of [38] to us.

Chapter 4

Sparse Boolean matrix multiplication

Let $\mathcal{O}(n^\omega)$ denote the time complexity of multiplying two $n \times n$ matrices. The classical algorithm as tought in school books requires a total of n^3 multiplications and $n^3 - n^2$ additions and thus gives a complexity bound of $\omega \leq 3$.

We can solve a system of linear equations, $AX = B$, by using the matrix multiplication $X = (A^{-1}B)$ or by using Gaussian Elimination. In 1965 Klyuvev et al. [55] showed that Gaussian elimination is optimal for solving systems of linear equations if only operations on rows and columns are allowed. However, in 1968 Winograd [99] modified algorithms for matrix multiplication and for solving systems of linear equations, thereby reduced the number of operations needed by a constant factor of two. In 1969 Strassen [92] was the first to discover a nontrivial algorithm that reduces the asymptotic upperbound of $\mathcal{O}(n^3)$ for matrix multiplication. He showed that a 2×2 matrix multiplication could be done using 7 instead of 8 multiplications, and by a recursive construction, a divide and conquer algorithm, an $n \times n$ matrix multiplication could be done using $\mathcal{O}(n^{\log 7}) \approx \mathcal{O}(n^{2.807})$ operations. That is, $\omega \leq 2.807$.

After Strassens' breakthrough it took around a decade before the upper bound was reduced again, this time by Pan [77] who reduced it to $\omega \leq 2.795$. With Pan's reduction a minor race on reducing the exponent began. See Figure 4.1 on the next page for a rough timeline of this decrement and [78] for a survey of the decreasing upper bound until 1981.

The current upper bound on ω is due to Coppersmith and Winograd [26] which in 1987 presented an algorithm for square matrix multiplication using $\mathcal{O}(n^{2.376})$ operations.

It is widely conjectured that the lower bound $\mathcal{O}(n^2)$ is also an upper bound on ω.

The above complexities are relevant for square matrices only. Copper-

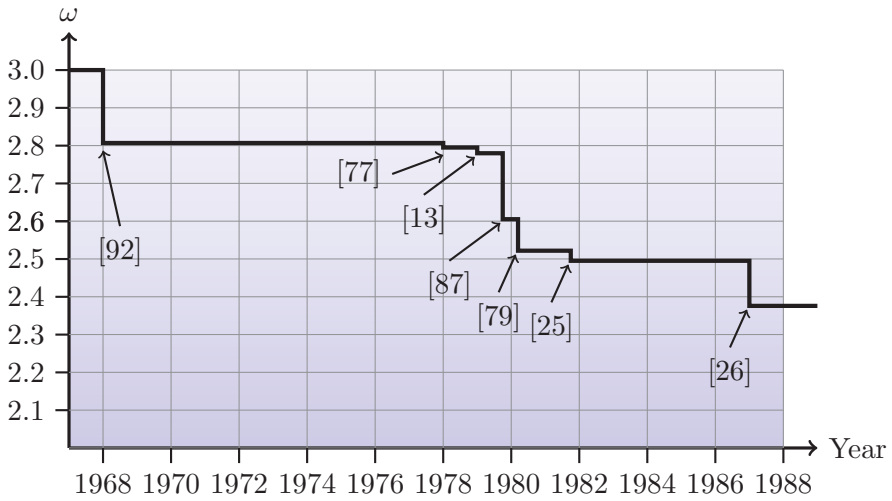


Figure 4.1: A sketch of the history of the decreasing bound on ω for square matrix multiplication.

smith [24] and Huang and Pan [50] extended the results to rectangular matrices. This resulted in a complexity bound for multiplying an $n \times \ell$ with an $\ell \times n$ matrix of $\mathcal{O}(n^{1.843+o(1)}\ell^{0.533} + n^{2+o(1)})$, assuming $\omega = 2.376$.

In 2008 Coppersmith and Winograd [27] showed that given one algorithm for multiplying matrices, there exists another, better, algorithm. As a consequence, ω is a limit point and can therefore not be realized by any single algorithm.

All algorithms above use only the side lengths of the matrices as parameter. In 2005 Yuster and Zwick [104] presented an algorithm, that also considers the number N of non-zeros in the matrices. In database terms, where matrices are often affinity matrices representing the input data, N corresponds to the input size. Their algorithm had a complexity of $\mathcal{O}(N^{0.7}n^{1.2}+n^{2+o(1)})$, implying that for $N \leq n^{1.14}$ their algorithm is almost optimal, using only $\mathcal{O}(n^{2+o(1)})$ operations. The essence of their approach was to split the data into two subsets, determined by a function of n and N, and to produce the output by applying the fast dense matrix multiplication by Coppersmith and Winograd [26] and the naive sparse matrix multiplication (see [43]) to the respective sets. The latter algorithm basically omits considering elements from the two matrices if both elements are zero.

The content of this chapter is based on our paper [8] that extends the idea of Yuster and Zwick by also considering the output size Z, ie. the number of non-zeros in the result matrix, as a parameter. We thereby obtain a complexity upper bound of $\mathcal{O}(N^{0.862}Z^{0.408} + N^{2/3}Z^{2/3})$ in the RAM model, and $\mathcal{O}\left(\frac{N\sqrt{Z}}{BM^{1/8}}\right)$ in the I/O model. In Figure 4.3 on page 32 we compare our bound in the RAM model with the bounds of Coppersmith and Winograd

as well as Yuster and Zwick over the set of all possible input and output sizes.

The latest result is due to Lingas [60] who presented a randomized algorithm running in $\mathcal{O}(n^2 Z^{0.188})$ time.

4.1 Introduction

Efficient computation of matrix multiplication and joins of database relations are both problems that have been studied in decades. What might not be obvious is that the two problems are related and below we shall see how computation time of both can be improved in some cases by combining techniques from the fields.

First, let us spend a moment motivating the need for improvements in computation of database joins—or rather, database joins followed by a duplicate eliminating projection: consider a set of movies and the set of actors in these movies. This data set can be described in a table with two columns, *movie* and *actor*, pairing related movies and actors. If we were interested in the unique set of actor pairs playing together in at least one movie we could join the table with itself on *movie*, project away the *movie* column and eliminate duplicates. In a small experiment on a subset of the Internet Movie Database (IMDB) we performed a join of 492,000 movie appearances, involving 37,000 actors and 8,100 movies. The number of actor pairs produced by the join-project was 70,000,000, while the size of the join (without projection) was much larger, having over 676,000,000 tuples.

As a more generel formulation, consider two tables $R_1(a, b)$ and $R_2(b, c)$ sharing the key b, and a join on b followed by a projection on a and c. In relational algebra this can be written as $\pi_{a,c}(R_1 \bowtie R_2)$. It is easy to see that an algorithm for this case can be used to solve the case where the relations may have more attributes, by considering several attributes as one (if needed, hashing can be used to produce a unique signature for large composite values). We will use N and Z to denote the input and output size, respectively. That is, $N = |R_1| + |R_2|$ and $Z = |\pi_{a,c}(R_1 \bowtie R_2)|$. As an example, assume $R_1 = R_2 = \{(x, y) \in \mathbb{N}^2 \mid 1 \leq x \leq n \text{ and } 1 \leq y \leq n\}$ so that $|R_1| = |R_2| = n^2$ for some $n \in \mathbb{N}$. Current database systems produce the final result by evaluating the operators in an evaluation tree. We will refer to this approach as the *classical algorithm* (see [102]). In our case, this implies two steps: First, the join $R_1 \bowtie R_2$ is performed, producing an intermediate result of a certain size. Next, the projection is carried out. In the join $R_1 \bowtie R_2$, each of the n^2 tuples in R_1 will match n tuples in R_2 resulting in n^3 unique tuples in total for the join operation. However, when performing the projection $\pi_{a,c}$ afterwards, the final result will only have n^2 unique tuples when duplicates are eliminated. In other words, the intermediate result had a factor $\Theta(\sqrt{N})$ tuples more than both the input

and the final result which seems like a waste of costly I/O. Other cases are less trivial.

Let M and B denote the memory and block size respectively where the unit of measurement is a single relation entry. That is, we assume that the memory can hold M entries of a relation and a block can hold B entries [3]. Let furthermore $\tilde{\mathcal{O}}(f)$ be a shorthand for $f^{1+o(1)}$. Then the classical algorithm requires $\tilde{\mathcal{O}}(N\sqrt{Z}/B)$ I/Os (see Section 4.2). If the join attribute is not projected away the classical algorithm is good, running in $\tilde{\mathcal{O}}((N+Z)/B)$ I/Os.

As explained in more detail later, one way to improve the worst-case behavior in cases similar to the example above is to represent the input tuples of R_1 and R_2 as adjacency matrices of size $n \times n$ and construct the result by multiplying the matrices in $\tilde{\mathcal{O}}(n^{2.376})$ time [26].

This chapter presents a way to evaluate these kind of expressions more efficiently, without the need for the large intermediate subresult, by using a hybrid of matrix multiplication and the classical algorithm. The hybrid technique implies a worst-case improvement in the computation time of conventional sparse matrix multiplications where both input and output is sparse. The improvement holds within the RAM model and the I/O model [3]. More specifically, we obtain a worst-case time complexity of $\tilde{\mathcal{O}}(N^{2/3}Z^{2/3} + N^{0.862}Z^{0.408})$ in the RAM model and $\tilde{\mathcal{O}}\left(\frac{N\sqrt{Z}}{BM^{1/8}}\right)$ I/Os in the I/O model where, as a side effect of the hybrid construction, our algorithm is at least as good in worst-case as any known algorithm for matrix multiplication for all possible combinations of N, n and Z.

We will refer to joins followed by a duplicate eliminating projection that projects away one or more join attributes as a *collapsing join-project*. Potentially, collapsing join-projects can be used as a single operator in query optimizers.

4.1.1 Related work

In 1984 Willard [96] presented an algorithm for evaluation of relational calculus expressions and analyzed the worst-case complexity in the RAM model. The shown time and space bounds had only the input size N as parameter. In 1990 Willard [97] presented an improved analysis which also took the output size Z into account. Willard did not consider projections and was therefore able to achive near-linear complexity in his algorithms. However, the results did not scale in the number of tables k used. Pagh and Pagh [74] introduced k as a third parameter in their analysis of acyclic joins and presented an algorithm that scales linearly with k. As seen, the analysis can be more precise when using more parameters. This chapter considers k in Section 4.2 but we focus on the case $k = 2$ in our algorithm in Section 4.3. However, we introduce a fourth parameter, namely the number n of distinct

attribute values in input.

We will refer to our generic algorithm as *Algorithm 2*. The generic algorithm has a number of instantiations, depending on how its steps are implemented (in the RAM or I/O model). Below, we compare the worst-case performance *analysis* of Algorithm 2 in the RAM model with the *analysis* of the classical sort-merge-join, the results by Coppersmith and Winograd [26] and Yuster and Zwick [104]. We emphasize *analysis* because the various analyses are not tight to the actual performance of the algorithms. The shown comparison is therefore not accurate. In the following, let n denote then number of distinct attribute values in the input, that is, $n = |\pi_a(R_1) \cup \pi_b(R_1) \cup \pi_b(R_2) \cup \pi_c(R_2)|$.

Algorithm 2 The analysis of this algorithm gives a complexity of

$$\tilde{\mathcal{O}}(N^{2/3}Z^{2/3} + N^{0.862}Z^{0.408}).$$

The classical algorithm Yannakakis [102] gave a worst-case complexity of $\tilde{\mathcal{O}}(NZ)$ for general acyclic join-projects on an arbitrary number of relations. For two relations, the worst-case complexity is $\tilde{\mathcal{O}}(N\sqrt{Z})$ as we show in Theorem 4.1. This anaysis is tight.

Coppersmith and Winograd We will refer to this result as *CW*. They obtained a matrix exponent of 2.376 giving a complexity of $\tilde{\mathcal{O}}(n^{2.376})$. This analysis is tight.

Yuster and Zwick We will refer to this result as *YZ*. Their complexity was $\tilde{\mathcal{O}}(N^{0.7}n^{1.2} + n^2)$ for $n \times n$ matrices with at most N nonzero elements but the analysis is not output sensitive. Notice that $n \leq N$.

The space requirements for the above algorithms are generally determined by the size of the intermediate results and the size of the matrices involved.

Table 4.1 compares the time and space requirements for the above algorithms and in Figure 4.2 we show, for each $(N, Z) \in [n^1; n^2] \times [0; n^2]$, the fastest algorithm (excluding Algorithm 2) at that coordinate with respect to their analysis. Figure 4.3 shows where the analysis of Algorithm 2 is (strictly) best.

4.1.2 Outline

The rest of this chapter is organized as follows: above we gave an example of suboptimal behavior of the classical algorithm and this behavior will be analyzed more formally in Section 4.2. Section 4.3 describes our algorithm in the RAM and I/O model.

Algorithm	Model	Time	Space
Classical alg.	RAM	$\tilde{\mathcal{O}}(N\sqrt{Z})$	$\tilde{\mathcal{O}}((N+Z)/w)$
Algorithm 2	RAM	$\tilde{\mathcal{O}}(N^{2/3}Z^{2/3} + N^{0.862}Z^{0.408})$	$\tilde{\mathcal{O}}(T/w)$
CW	RAM	$\tilde{\mathcal{O}}(n^{2.376})$	$\tilde{\mathcal{O}}(n^2/w)$
YZ	RAM	$\tilde{\mathcal{O}}(N^{0.7}n^{1.2} + n^2)$	$\tilde{\mathcal{O}}(T/w)$
Classical alg.	I/O	$\tilde{\mathcal{O}}(N\sqrt{Z}/B)$	T
Algorithm 2	I/O	$\tilde{\mathcal{O}}\left(\frac{N\sqrt{Z}}{BM^{1/8}}\right)$	T

Table 4.1: A comparison of worst-case time and space requirements for the algorithms mentioned in Section 4.1.1. The units for time and space in the RAM model are *steps* and *words* of size w, respectively, and in the I/O model, the units are number of I/Os and number of blocks of size B, respectively. T is a short-hand notation for the time complexity of the algorithm on the *same* line.

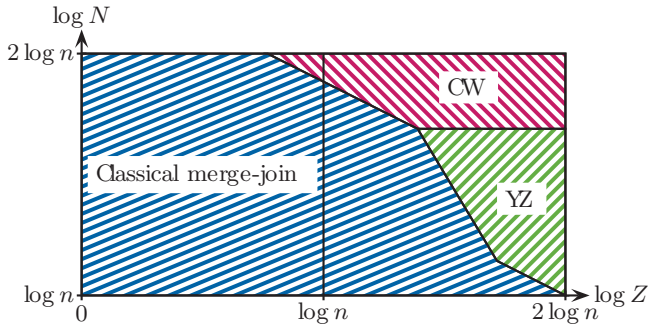


Figure 4.2: A comparison of the classical merge-join algorithm and the algorithms by Coppersmith and Winograd (CW) and Yuster and Zwick (YZ). The figure shows the previously fastest algorithm on a RAM model for different values of parameters N (input size) and Z (output size).

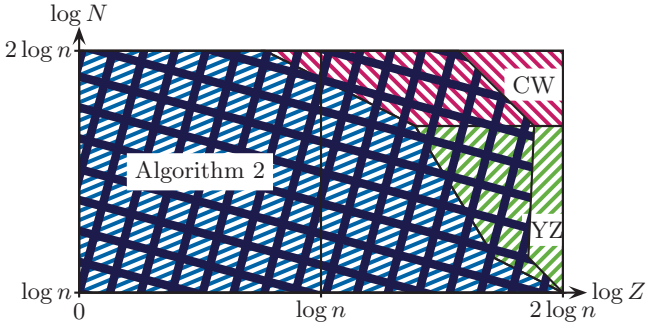


Figure 4.3: A comparison similar to Figure 4.2 but with the analysis of Algorithm 2 (the area under the grid) included. The graph shows the strictly fastest algorithm on a RAM model for different values of parameters N and Z. As seen, the analysis of Algorithm 2 completely dominates the merge-join and for some values of (N, Z) it also dominates the algorithms by Coppersmith and Winograd, and Yuster and Zwick.

4.2 The classical algorithm

In this section we perform an analysis of the classical algorith. Let $\bowtie R_i$ denote a *natural join* of k relations $R_1, \ldots, R_k$ and Z denote the output size of the final projection $\pi(\bowtie R_i)$. Given a known input size $N = \sum |R_i|$ and output size Z we search for an upper bound for

$$\mathcal{U}(N, Z) = \max_{\substack{R_1 \ldots R_k \\ \Sigma |R_i| = N \\ |\pi(\bowtie R_i)| = Z}} |\bowtie R_i|.$$

In 1981 Yannakakis [102] showed that $\mathcal{U}(N, Z) \leq NZ$ by analyzing an algorithm that is identical to the classical algorithm when all relations share an attribute. But $\mathcal{U}$ depends on k as the following theorem shows. From now on we consider the case where all relations share an attribute.

Theorem 4.1 *Let $k > 1$ be an integer. For k relations on the form $R_i(a_i, b)$ we have*

$$\mathcal{U}(N, Z) = \Theta(NZ^{1-\frac{1}{k}}).$$

Proof We first show the upper bound on $\mathcal{U}$. For each possible b-value x, define $s_i(x)$ as the number of tuples in R_i having $b = x$. That is, $s_i(x) = |\sigma_{b=x}(R_i)|$. The tuples in $\bowtie R_i$ having $b = x$ for some value x will all be unique and thus have a representative in the final projected output. Therefore

$$s_1(x)s_2(x) \cdots s_k(x) \leq Z. \tag{4.1}$$

For any i, define S_i as the subset of b-values occurring in more than $Z^{\frac{1}{k}}$ tuples of R_i, that is $S_i = \{x \mid s_i(x) > Z^{\frac{1}{k}}\}$. Each $x \in S_i$ will match at most $Z^{1-\frac{1}{k}}$ tuples in total in the other tables due to (4.1), and as $|S_i| \leq |R_i|$ we have that $x \in S_i$ will induce at most $|R_i|Z^{1-\frac{1}{k}}$ tuples in the final projected output. A similar argument can be applied for all i resulting in

$$\mathcal{U}(N, Z) \leq \sum_{i=1}^{k} |R_i| Z^{1-\frac{1}{k}} = NZ^{1-\frac{1}{k}}.$$

For the lower bound of $\mathcal{U}$ let $[q]$ be a general notation for the set $\{x \in \mathbb{N} \mid 1 \leq x \leq q\}$ and define k relations $R_i(a_i, b)$ with tuples $[Z^{\frac{1}{k}}] \times [\frac{N}{k}/Z^{\frac{1}{k}}]$. Note that $|R_i| = \frac{N}{k}$ and that every tuple $r \in R_1$ will match exactly $Z^{\frac{1}{k}}$ tuples in each of the $k-1$ other relations producing a total of $(Z^{\frac{1}{k}})^{k-1} = Z^{1-\frac{1}{k}}$ tuples in the join containing r. As $|R_1| = \frac{N}{k}$ the total join size is $\frac{N}{k}Z^{1-\frac{1}{k}}$. □

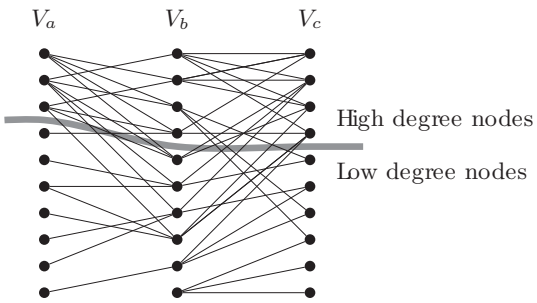


Figure 4.4: Two relations R_1 and R_2 represented as a graph.

4.3 Computing the join-project

We will show how to compute the collapsing join-project efficiently for $k = 2$. For $R_1(a, b)$ and $R_2(b, c)$ let V_a, V_b and V_c be sets of all distinct a, b and c values represented in such a way that $v \in V_i$ and $u \in V_j$ where $u = v$ are treated as equal if $i = j$ but distinct if $i \neq j$. The join $R_1 \bowtie R_2$ can be represented as a *sequentially tripartite* graph $G = (V_a, V_b, V_c, E)$ where $E \subseteq (V_a \times V_b) \cup (V_b \times V_c)$. Notice, that in contrast to a conventional tripartite graph we have that $V_a \times V_c \cap E = \emptyset$. We consider undirected graphs, where it is understood that an edge (u, v) is considered identical to the edge (v, u).

Let $(v_a, v_b) \in V_a \times V_b$ be an edge in E if and only if (v_a, v_b) is a tuple in $R_1(a, b)$ and similarly $(v_b, v_c) \in V_b \times V_c$ an edge in E if and only if (v_b, v_c) is a tuple in $R_2(b, c)$. See Figure 4.4. Notice that (a, c) is a tuple in $\pi_{a,c}(R_1 \bowtie R_2)$ if and only if there is a path of length 2 from a to c in G. The edges in $V_a \times V_b$ and $V_b \times V_c$ can be represented as two *adjacency matrices* M^{ab} and M^{bc}. We have the following easy lemma:

Lemma 4.2 *A tuple $(a, c) \in \pi_{ac}(R_1 \bowtie R_2)$ if and only if $(M^{ab}M^{bc})_{a,c} > 0$.*

Proof As M^{ab} and M^{bc} are adjacency matrices over a graph their product will, by definition of matrix multiplication, contain a non-zero entry at row a column c exactly if a and c are connected by a path of length 2. □

Rather than just using matrix multiplication we will compute the result by decomposing the join and projection into several parts, defined by how much redundancy they are candidate to produce in the classical algorithm.

Definition 4.3 (Degree) *Let $\delta(v) : V \to \mathbb{N}$ denote the* degree *of the node $v \in V = V_a \cup V_b \cup V_c$ defined as the size of $(\{v\} \times V) \cap E$.*

Using the degree, we can give a simple upper bound on the number of occurrences of every tuple (a, c) in the join $R_1 \bowtie R_2$:

Lemma 4.4 *Let $B_1(a) = \{b \in V_b \mid (a, b) \in E\}$ and $B_2(c) = \{b \in V_b \mid (b, c) \in E\}$. Then the a tuple of the form $(a, \cdot, c)$ will occur exactly $r = |B_1(a) \cap B_2(c)|$ times in the join $R_1 \bowtie R_2$ and $r \leq \min(\delta(a), \delta(c))$.*

We will split the nodes in V_a, V_b and V_c in low and high degree nodes using two thresholds $\Delta_{ac}, \Delta_b \in \mathbb{N}$. According to Lemma 4.4 the values from V_a and V_c with a degree smaller than Δ_{ac} are guaranteed to occur with multiplicity at most Δ_{ac} in the join. Tuples in R_1 and R_2 containing such a and c values can therefore be joined by using a conventional merge join and removing duplicates using either a dictionary or by sorting, depending on model of computation. The output multiplicity cannot be deduced from the degree of values in V_b but tuples having $\delta(b) < \Delta_b$ will occur at most $N\Delta_b$ times in the join and handling these small-degree tuples by a merge-join will imply a smaller input to the following more time-consuming step: The rest of the tuples are represented as two adjacency matrices which are multiplied using an efficient conventional matrix multiplication algorithm [104] in order to find paths of length 2 between a and c nodes. The algorithm is summarized in Algorithm 2.

Observation 4.5 *Conventional matrix multiplication is a special case of Algorithm 2 for $\Delta_{ac} = \Delta_b = 0$. The algorithm by Yuster and Zwick [104] is a special case of Algorithm 2 for $\Delta_{ac} = n + 1$. The classical merge-join algorithm is a special case of Algorithm 2 for $\Delta_{ac} = \Delta_b = n + 1$.*

In particular, there exist values Δ_{ac} and Δ_b so that Algorithm 2 is at least as good as any other known algorithm for sparse Boolean matrix multiplication.

Also notice, that the algorithm will produce the correct output (but the running times may differ) for any values of Δ_{ac} and Δ_b.

Algorithm 2 Computing $\pi_{ac}(R_1 \bowtie R_2)$ or equivalently: computing the product of M^{ab} and M^{bc}.

1: $\mathcal{R}_1^{'} \leftarrow \{(a,b) \in R_1 \mid \delta(a) < \Delta_{ac}\}$ ▷ Low multiplicity output
2: $\mathcal{R}_2^{'} \leftarrow \{(b,c) \in R_2 \mid \delta(c) < \Delta_{ac}\}$
3: $S \leftarrow \pi_{ac}(\mathcal{R}_1^{'} \bowtie \mathcal{R}_2)$ using the classical algorithm
4: $S \leftarrow S \cup \pi_{ac}(\mathcal{R}_1 \bowtie \mathcal{R}_2^{'})$ using the classical algorithm
5: $\mathcal{R}_1^{''} \leftarrow \{(a,b) \in R_1 \mid \delta(b) < \Delta_b\}$ ▷ Low $\delta(b)$
6: $\mathcal{R}_2^{''} \leftarrow \{(b,c) \in R_2 \mid \delta(b) < \Delta_b\}$
7: $S \leftarrow S \cup \pi_{ac}(\mathcal{R}_1^{''} \bowtie \mathcal{R}_2^{''})$ using the classical algorithm
8: $M^{'} \leftarrow$ adjacency matrix for $\{(a,b) \in R_1 \mid \delta(a) \geq \Delta_{ac}$ and $\delta(b) \geq \Delta_b\}$
9: $M^{''} \leftarrow$ adjacency matrix for $\{(b,c) \in R_2 \mid \delta(c) \geq \Delta_{ac}$ and $\delta(b) \geq \Delta_b\}$
10: $\mathcal{M} \leftarrow M^{'}M^{''}$ using multiplication algorithm of choice
11: $S \leftarrow S \cup \{(a,c) \mid \mathcal{M}_{a,c} > 0\}$
12: Eliminate duplicates in S
13: Output S

4.3.1 Complexity in the RAM model

In the following we assume that Z is known. This assumption will be justified later.

Theorem 4.6 *Let $f(N, Z)$ denote the time complexity of Algorithm 2. Then $f(N, Z)$ is $\tilde{\mathcal{O}}(N^{2/3}Z^{2/3} + N^{0.862}Z^{0.408})$ for suitable choice of Δ_{ac} and Δ_b.*

Proof Algorithm 2 produces the output in three steps that account for the superlinear work with respect to the $\tilde{\mathcal{O}}$ notation:

1. Tuples generated in line 3 and 4: There are Z unique tuples in $\pi_{ac}(R_1 \bowtie R_2)$ and each tuple corresponds to at most Δ_{ac} tuples in $R_1 \bowtie R_2$ according to Lemma 4.4 so this step produces at most $O(Z\Delta_{ac})$ tuples in total.

2. Tuples generated in line 7: Each b node can be reached from at most Δ_b different a or c nodes. Therefore this step contributes with at most $O(N\Delta_b)$ tuples.

3. Tuples generated in line 10 and 11: The size of the matrices will be at most[1] $\frac{N}{\Delta_{ac}} \times \frac{N}{\Delta_b}$ and $\frac{N}{\Delta_b} \times \frac{N}{\Delta_{ac}}$ so this step can be handled in $\tilde{\mathcal{O}}(M(\frac{N}{\Delta_{ac}}, \frac{N}{\Delta_b}, \frac{N}{\Delta_{ac}}))$ time where $M(x, y, z)$ denotes the minimum number of arithmetic operations needed in order to multiply an $x \times y$ with a $y \times z$ matrix. This can be implemented in $\tilde{\mathcal{O}}(M(x, y, z))$ time in the RAM model.

Huang and Pan [50] proved that

$$M(x, y, x) = x^{2-\alpha\beta+o(1)}y^{\beta} + x^{2+o(1)}$$

is an upper bound on M, where α and β are constants given by the matrix multiplication algorithm. With bounds proved by Coppersmith and Winograd [26] and Coppersmith [24] the currently best known algorithm has $\alpha = 0.294$ and $\beta = 0.533$.

The duplicate elimination in line 12 can be done in $\tilde{\mathcal{O}}(Z)$ time using sorting or hashing.

[1] The matrix dimensions are also bounded by n but as seen in figure 4.3 we still obtain a near-optimal result by simplifying the analysis.

We now have

$$\begin{aligned} f(N,Z) &= \tilde{\mathcal{O}}\left(M(\tfrac{N}{\Delta_{ac}},\tfrac{N}{\Delta_b},\tfrac{N}{\Delta_{ac}}) + N\Delta_b + Z\Delta_{ac}\right) \\ &= \tilde{\mathcal{O}}\Big((\tfrac{N}{\Delta_{ac}})^{2-\alpha\beta+o(1)}(\tfrac{N}{\Delta_b})^{\beta} + (\tfrac{N}{\Delta_{ac}})^{2+o(1)} \\ &\qquad + N\Delta_b + Z\Delta_{ac}\Big) \qquad (4.2) \\ &= \tilde{\mathcal{O}}\Big((\tfrac{N}{\Delta_{ac}})^{k}(\tfrac{N}{\Delta_b})^{\beta} + (\tfrac{N}{\Delta_{ac}})^{2} \\ &\qquad + N\Delta_b + Z\Delta_{ac}\Big). \qquad (4.3) \end{aligned}$$

where (4.3) is a simplified expression obtained by setting $k = 2 - \alpha\beta$.

We are interested in values for Δ_{ac} and Δ_b so that $f(N,Z)$ is minimized. For simplicity, rewrite (4.3) to $\max\{\dots\}$ of the involved terms

$$f(N,Z) = \tilde{\mathcal{O}}\left(\max\left\{(\tfrac{N}{\Delta_{ac}})^{k}(\tfrac{N}{\Delta_b})^{\beta}, (\tfrac{N}{\Delta_{ac}})^{2}, N\Delta_b, Z\Delta_{ac}\right\}\right).$$

It is now safe to assume that $N\Delta_b = Z\Delta_{ac}$ and therefore we can simplify the above equation by setting $\Delta_b = Z\Delta_{ac}/N$:

$$f(N,Z) = \tilde{\mathcal{O}}\left(\max\left\{(\tfrac{N}{\Delta_{ac}})^{k}(\tfrac{N^2}{Z\Delta_{ac}})^{\beta}, (\tfrac{N}{\Delta_{ac}})^{2}, Z\Delta_{ac}\right\}\right)$$

Notice that the two first parameters decrease with Δ_{ac} while the last one increases. This means that minimum exists where either $(\frac{N}{\Delta_{ac}})^{k}(\frac{N^2}{Z\Delta_{ac}})^{\beta} = Z\Delta_{ac}$ or $(\frac{N}{\Delta_{ac}})^{2} = Z\Delta_{ac}$.

In order to deduce Δ_{ac}, consider the first case:

$$\begin{aligned} (\tfrac{N}{\Delta_{ac}})^{k}(\tfrac{N^2}{Z\Delta_{ac}})^{\beta} = Z\Delta_{ac} \quad &\Rightarrow \\ \Delta_{ac} = N^{\frac{k+2\beta}{1+k+\beta}} Z^{\frac{-2}{1+k+\beta}}&. \end{aligned}$$

With this value of Δ_{ac} we obtain the minimum

$$\tilde{\mathcal{O}}(Z\Delta_{ac}) = \tilde{\mathcal{O}}\left(N^{\frac{k+2\beta}{1+k+\beta}} Z^{1-\frac{2}{1+k+\beta}}\right).$$

Similarly, for the second case where $(\frac{N}{\Delta_{ac}})^2$ dominates we have the minimum

$$\tilde{\mathcal{O}}(Z\Delta_{ac}) = \tilde{\mathcal{O}}(N^{2/3}Z^{2/3})$$

using $\Delta_{ac} = N^{2/3}/Z^{1/3}$.

Finally the sum

$$\begin{aligned} f(N,Z) &= \tilde{\mathcal{O}}\left(N^{\frac{k+2\beta}{1+k+\beta}} Z^{1-\frac{2}{1+k+\beta}} + N^{2/3}Z^{2/3}\right) \\ &\approx \tilde{\mathcal{O}}(N^{0.862}Z^{0.408} + N^{2/3}Z^{2/3}). \qquad (4.4) \end{aligned}$$

must be an upper bound for the minimum, where the last line is obtained by using the currently best values of $\alpha = 0.294$ and $\beta = 0.533$ as described above. □

Observation 4.7 *If the exponent in matrix multiplication is* $2+o(1)$ *as conjectured by many, the worst-case complexity of Algorithm 2 is* $\tilde{\mathcal{O}}(N^{2/3}Z^{2/3})$ *for suitable values of* Δ_{ac} *and* Δ_b*.*

Notice, that the above observation implies that $O(N^{4/3})$ is a lower bound for the complexity, even if the output has size $\mathcal{O}(N)$.

As noted in the proof of Theorem 4.6 our analysis is simplified and our choice of Δ_{ac} and Δ_b not optimal: we do not take into account that $n \times n$ is an upper bound on matrix sizes and $\Delta_{ac}, \Delta_b \leq n$. The real optimum is found by minimizing

$$\begin{aligned}
&\tilde{\mathcal{O}}\Big(M\left(\min(\tfrac{N}{\Delta_{ac}}, n), \min(\tfrac{N}{\Delta_b}, n), \min(\tfrac{N}{\Delta_{ac}}, n)\right) \\
&\qquad + N\Delta_b + Z\Delta_{ac}\Big) \\
&= \tilde{\mathcal{O}}\Big(M\left(\min(\tfrac{N}{\Delta_{ac}}, n), \min(\tfrac{N^2}{Z\Delta_{ac}}, n), \min(\tfrac{N}{\Delta_{ac}}, n)\right) \\
&\qquad + Z\Delta_{ac}\Big)
\end{aligned}$$

for $\Delta_{ac} \leq n$.

Observation 4.8 *The optimal values of* Δ_{ac} *and* Δ_b *can be found efficiently assuming* Z *is known.*

4.3.2 Output sensitivity

When executing the algorithm, Z is not known in advance but it can be found iteratively without altering the complexity of the algorithm. This is done by iteratively guessing a value of $Z \in [Z'; 2Z'[$ for $Z' = 2^i$ in iteration i and noticing that the algorithm still works correctly when using an upper bound of Z. In each iteration the algorithm is stopped when the execution time exceeds the bound described in Theorem 4.6. As the execution time decreases geometrically, the latest execution time will dominate.

The time bound above, however, is given in big-oh notation which makes it impossible in practice to compare the actual evaluation time with the theoretical bound. A practical comparison would require an analysis of the involved constant. Another approach could be to estimate the value of Z by sampling: let S denote a sample obtained by picking q nodes from $\pi_a(R_1)$, q nodes from $\pi_c(R_2)$ and computing all paths of length 2 between these nodes. This sample S would have an expected size $(q/N)^2 Z$ and thus $Z = \mathbf{E}[(N/q)^2|S|]$, i.e. we have an unbiased estimator for Z.

4.3.3 Complexity in the I/O model

We can obtain results analogous to those in Section 4.3.1 for the I/O model by using I/O efficient algorithms for the steps of Algorithm 2, including an I/O efficient version of fast matrix multiplication. However, as we will see below even a very simple matrix multiplication algorithm, a cache-aware version of the cubic algorithm, yields worst-case complexity better than the classical algorithm.

Matrix multiplications in the I/O model can be performed by grouping the matrix elements in squares of size $\sqrt{M} \times \sqrt{M}$ and performing a conventional matrix multiplication using these squares as element units. With a block size of B, such a matrix multiplication requires $(N/\sqrt{M})^3 \frac{M}{B} = N^3/(B\sqrt{M})$ I/Os.

Theorem 4.9 *The number of I/Os required by Algorithm 2 when using a cache-aware cubic matrix multiplication algorithm is* $\tilde{\mathcal{O}}\left(\frac{N\sqrt{Z}}{BM^{1/8}}\right)$.

Proof Consider the three steps mentioned in the proof of Theorem 4.6. Step 1 requires $\tilde{\mathcal{O}}(Z\Delta_{ac}/B)$ I/Os and step 2 requires $\tilde{\mathcal{O}}(N\Delta_b/B)$ I/Os using sorting to compute the join and eliminate duplicates. We use the simple cubic-time matrix multiplication algorithm described above for step 3 resulting in a requirement of $N^3/(\Delta_{ac}^2\Delta_b\sqrt{M}B)$ I/Os for that step. Summarized, the number of I/Os required is

$$\tilde{\mathcal{O}}\left(\frac{Z\Delta_{ac}}{B} + \frac{N\Delta_b}{B} + \frac{N^3}{\Delta_{ac}^2\Delta_b\sqrt{M}B}\right). \tag{4.5}$$

Using similar arguments as in the proof of Theorem 4.6 we can assume that the three terms are equal at optimum. Setting the two first terms equal gives $\Delta_b = Z\Delta_{ac}/N$ which can be inserted into (4.5):

$$\tilde{\mathcal{O}}\left(2\frac{Z\Delta_{ac}}{B} + \frac{N^4}{\Delta_{ac}^3 Z\sqrt{M}B}\right) \tag{4.6}$$

Similarly, equating the two remaining terms gives $\Delta_{ac} = N/(\sqrt{Z}M^{1/8})$ which can be inserted into (4.6) in order to achive the desired result

$$\tilde{\mathcal{O}}\left(\frac{N\sqrt{Z}}{BM^{1/8}}\right) \text{ I/Os.}$$

□

Notice that even though we are using the naive cubic-time multiplication algorithm, this result improves the complexity with a factor $M^{1/8}$ compared to the classical algorithm.

4.4 Conclusion

We presented an output-sensitive algorithm for collapsing join-projects and sparse Boolean matrix multiplication that is more efficient worst-case in both the RAM and I/O model than currently known algorithms. As we only deal with worst-case analysis in this chapter the algorithm is not meant to replace current algorithms in database systems but might be implemented and used by the query optimizer as an alternative operator if the used join-project plan is slow. The presented algorithm can be modified to be used for conventional matrix multiplication over an arbitrary ring as well.

Chapter 5

Using the GPU

In the last two decades there has been a rapid evolution of graphics hardware that has made it relevant for non-graphical computation tasks, and a relatively new research field for algorithms and data structures on graphics hardware has therefore appeared.

The earliest example of a graphics processing unit (GPU) probably dates back to the 1970s where the ANTIC and CTIA chips provided a hardware assisted mix of graphics and text mode. The chips were simple and the use of graphics in commodity computers was limited. In the 1980s, the IBM Professional Graphics Controller was the first to provide hardware accelerated 2D and 3D graphics for IBM PCs. However, the hardware was slow (8 MHz), costed around $4500 USD and was therefore too expensive to succeed in the market. In the 1990s the graphics hardware started to evolve and be affordable to the average PC user. S3 Graphics introduced the first single-chip dedicated 2D accelerator named S386911 and many spin-off chips followed shortly thereafter with multiple APIs to access the hardware. Among those were Microsoft's WinG graphics library and DirectDraw. In the 1990s CPU-assisted 3D games became increasingly popular, which created a market for cheap dedicated 3D hardware. The first low-cost 3D chips were S3 ViRGE, ATI Rage and Matrox Mystique. At that time, two generalized cross-platform graphics APIs, OpenGL and Glide, competed to become the standard, and OpenGL won this battle. In the latest decade, from year 2000 and onward, the capacity of graphics hardware has increased rapidly with focus on being able to solve simple dedicated tasks very fast. Graphics hardware today is therefore characterized by its high processing power, high memory bandwidth and numerous computation cores. This is an ideal environment for data intensive or highly parallelizable computation tasks, and two popular APIs have been developed for utilizing the hardware for generalized non-graphics computation: OpenCL and CUDA. More information on the history of GPUs can be found in [59, 95].

In this chapter we use modern graphics hardware via OpenCL to perform

frequent pair mining, which is equivalent to Boolean matrix multiplication and join-projects. The chapter is an elaboration of the paper *A New Data Layout For Set Intersection on GPUs*, written in collaboration with Rasmus Pagh.

5.1 Introduction

Graphics processing units (GPUs) are currently the technology that gives the largest computing power per dollar (measured in floating-point operations per second) [35, 61, 69]. Developing algorithms for GPU computation is challenging, since the architecture imposes many requirements on the way algorithms work, if the potential is to be fully utilized. In particular, programs need to be structured in identical threads with as little conditional code as possible (i.e., having regular control flow), such that all threads can run the same instruction at the same time. Also, the memory access pattern of threads that execute together needs to be highly regular to approach the theoretical bandwidth of the GPU memory. For computation intensive tasks the availability of hundreds of processing units has resulted in large speedups compared to CPU computation (see e.g. the survey [73]). Even for data intensive tasks such as sorting, advantage over CPU computation has been demonstrated (see e.g. [42, 46, 57]).

Many computational problems depend on being able to perform set intersection efficiently. For example:

- in *Boolean matrix multiplication* of two matrices, M and M', we want to find all pairs (i, j) for which $\exists k : M_{i,k}M'_{k,j} > 0$, or equivalently for $A_i = \{j|M_{i,j} > 0\}$ and $B_j = \{i|M'_{i,j} > 0\}$, the pairs (i, j) for which $A_i \cap B_j \neq \emptyset$

- in a database context we might ask for a *join-project* of two tables, i.e., a join of two tables followed by a duplicate eliminating projection that projects away the join attribute. This is equivalent to sparse Boolean matrix multiplication [8], and thus dependent on efficient set intersection as well

- *frequent itemset mining* asks, given a set of transactions $T_1, \ldots, T_m$, where $T_i \subseteq \{1, \ldots, n\}$, to report all sets $S \subset \{1, \ldots, n\}$ having *support* at least s in the transactions. The support of S is defined as the number of transactions that have S as a subset. The special case where itemsets are limited to size two (where only item *pairs* are found) is also the core problem when larger itemsets are allowed, and frequent itemset mining in general therefore reduces to efficient set intersection

- all *conjuctive queries* can be thought of as set intersections: given a dataset D, and two pre-processed subsets of data, $f, g : D \to \{0, 1\}^{|D|}$,

the conjuctive query $\{d \in D | f(d) \wedge g(d)\}$ is exactly equivalent to an intersection.

In this chapter we consider the general problem of intersecting sets. However, we use frequent itemset mining as a case study throughout the text, as it is one of the most studied problems that can be solved by reduction to multiple set intersections. We furthermore focus on itemsets of size two (frequent pair mining), since this special case already has many applications (such as finding binary associations) and is highly challenging when there are many frequent items. We outline how our approach could be generalized to deal with larger itemsets.

Set representations in frequent itemset mining There are two principal ways of representing a set of transactions. In the standard *horizontal* format the transactions are stored one by one (possibly sorted), whereas the *vertical* format stores, for each item i, the set S_i of indices of transactions that contain i. This set is sometimes referred to as the *tidlist* of i. If S_i and S_j are sorted it is an easy task to compute the support of $\{i, j\}$ in time $\mathcal{O}(|S_i| + |S_j|)$. If the number of distinct items n is large we see that it is easy to parallelize the computation of all support counts: simply distribute the intersections among the processors such that each processor is responsible for support counts involving a small number of items.

For some data sets it is faster to use a horizontal layout and maintain a data structure that counts the occurrences of all pairs. Then the time spent on a pair $\{i, j\}$ is proportional to the support of $\{i, j\}$ rather than to the sum of support of $\{i\}$ and $\{j\}$. However, this approach may use excessive space when there are many pairs of frequent items. In parallel and distributed settings the high space usage translates into either using an expensive shared memory, or a phase where the support counts from different parts of the transactions are combined. In either case, the communication among processes becomes a bottleneck as the number of frequent items grows.

5.1.1 This chapter

Theoretical contribution We present a new data format for sets, BatMap, that is especially well-suited for parallel and pipelined computation. It is instructive to compare our format to *bitmaps*, which have previously been used to store the sets S_i, using one bit per transaction [36]. To compute the support of $\{i, j\}$ one needs to perform the bit-wise `AND` of the bitmaps encoding S_i and S_j, and count the number of 1s. This task parallelizes very well, as the bitmaps can be split into any desired number of pieces to be processed individually, and there is a low communication overhead in combining the counts. It is also very friendly to modern pipelined processor architectures,

since no conditional code is needed, avoiding the branch mispredictions that have haunted previous frequent itemset mining algorithms using "vertical data formats" and set intersections [82]. Finally, since data can be accessed sequentially, bitmaps make optimal use of cache and prefetching.

The BatMap maintains these advantages, while being more space-efficient on sparse sets. The space usage is in fact within a small factor of the information theoretical minimum for representing sets of a given size, which is the largest imaginable compression. That is, if S_i and S_j are represented using batmaps B_i and B_j we can compute the size of $S_i \cap S_j$ using a word-by-word comparison of B_i and B_j. In contrast to normal compressed representations of sparse bitmaps, the steps of this computation are completely fixed, and parallelize immediately. The name BatMap indicates the similarity to the functionality of a bitmap, and suggests that this is something that Bruce Wayne might use to mine associations between criminals and crimes.

We should mention a limitation of batmaps compared to bitmaps: the result of combining two batmaps is not a batmap, so it cannot directly support the intersection of more than two sets. Towards the end of the chapter we outline possible ways of dealing with this limitation.

Experiments In Section 5.4 we investigate the performance characteristics of our algorithm, and an implementation of Apriori [4] and FP-growth [45] for varying density and number of distinct items. We find that our algorithm scales well in the number of distinct items, in terms of both computation time and memory usage. In addition, the algorithm performs well for dense instances. We also perform experiments comparing batmaps on GPU with merging of sorted lists, a standard CPU-based algorithm for computing intersection size.

5.1.2 Previous work

Set intersection

The algorithm for intersecting two sorted lists is folklore. In the literature it has been extended in two main directions. The first is *adaptive* intersection procedures, that use fewer comparisons when there are compact witnesses for the intersection, see e.g. [29]. In the worst case, and in the average case, these algorithms provide no speedup over the classical algorithm. Second, for dense sets there has been considerable work on compressed representations, usually referred to as *compressed bitmaps*. The *density* of a set is its size divided by the size of the universe from which its elements come (e.g., in the case of frequent itemset mining, the density of S_i is $|S_i|/m$). Previous work on high-performance compressed bitmap formats include Boncz [107], BBC [52] and WAH [101]. These methods all require data to be decoded

sequentially, and provide no easy parallelization.

Bille et al. [12] present a compressed bitmap format that is nearly optimal wrt. the amount of data read to compute set operations. However, this is mainly a theoretical result that is not likely to perform well in a GPU setting. Our new vertical data layout can be viewed as a kind of compressed bitmap, with special properties.

Frequent itemset mining

To ease the exposition we will assume that we have preprocessed the data set to remove items with support below the threshold we are interested in. All existing frequent itemset methods do this, in one way or another, so the interesting comparison is for the case where there are only frequent items.

GPU computation The previous work most closely related to ours is that of Fang et al. [36]. They use (in the PBI-GPU algorithm) a bitmap to store a vertical representation of the data set. This means that the representation of a data set of m transactions with n distinct items requires mn bits of space. For a sparse data set with a total of mb items, where $b \ll n$, this can be much more than the $\log \binom{mn}{mb} \approx mb \log(n/b)$ bits needed to represent the data. Experiments in [36], on hardware similar to what we use, show that their GPU/bitmap is more than 1 order of magnitude faster than a tuned implementation of the Apriori algorithm in some cases where the data set is dense (density 49%). For a sparse data set (density 0.6%) there is basically no speedup. So both from a space usage and a computation time perspective this method does not work well for sparse data sets. Based on the experimental results on the synthetic dataset T40I10D100K reported in [36] we can estimate the speed of the underlying set intersections to be around 40 Gbit per second. In the case of T40I10D100K, which has a density of 4%, this means that they can in 1 second intersect sets of total size around $1.6 \cdot 10^9$. Sets with lower density take proportionally longer per item, and sets with larger density take proportionally less time.

We also that [36] did not present experiments showing that a GPU implementation can be faster than FP-growth [45] (in fact, in all three experiments reported, FP-growth was considerably faster).

CPU computation A lot of work has been devoted to parallel and distributed implementations of frequent pattern mining. The survey of Zaki [106] describes the state-of-the-art as of 1999. More recent work has focused on multi-core architectures of modern commodity hardware, trying to optimize cache performance and minimize the overhead of access to shared data [39, 58]. However, GPU parallelism involves many constraints on the structure of the code and memory access pattern that is not addressed in

these works. In particular, our method exploits the massive SIMD parallelism that is available on GPUs, and we find it conceivable that the set representation we describe could lead to other advances in parallel and distributed computation.

5.2 BatMaps

Let S_i denote the set of transactions containing item i. We wish to preprocess the sets $S_i \subseteq \{1, \ldots, m\}$ such that we can quickly compute the intersection sizes $|S_i \cap S_j|$ for all item pairs $\{i, j\}$. A standard solution to this problem is to store the sets as sorted lists, which allows an intersection to be computed in time $\mathcal{O}(|S_i| + |S_j|)$ by simple merging. However, the control flow for this intersection procedure is unpredictable, which makes it work poorly on modern architectures, in particular GPUs, since they require highly structured control flow to perform well.

The initial idea is to rely on hashing rather than comparisons. If we organize the sets in hash tables (say, using linear probing or perfect hashing) it is indeed fast to determine the common elements of two sets S_i, S_j as we simply look up all elements from S_i in S_j. Using perfect hashing (perhaps with vectorization [14]) the control flow becomes deterministic and predictable. However, the memory access pattern of hash table lookups remains random and highly irregular.

Our new approach starts with an old idea from parallel and distributed data structures [30, 90, 94], applied in a novel way. The idea is to store sets redundantly to enable more efficient parallel/distributed operations — here we consider only the case where elements are stored in positions given by $2d - 1$ random hash functions. If an element is stored in d out of the $2d - 1$ possible positions, then for any two sets that both contain x there is at least one position that contains x in both representations. This means that it suffices to do a data independent element-by-element comparison which parallelizes very well (see Figure 5.1).

Our adaptation We will consider $d = 2$ and store each element $x \in S_i$ in two of three hash tables. For the time being we will simply think of these hash tables as a $3 \times r$ array $A^{(i)}$ (section 5.3 describes the specific layout we use). In each hash table $t \in \{1, 2, 3\}$ there is exactly one position $(t, h_t^{(i)}(x))$ where x can be stored, given by a hash function $h_t^{(i)}$. Figure 5.2 illustrates this idea.

It will be important that all sets are stored according to the *same* hash functions h_1, h_2, h_3, with range scaled according to the size of the set. That is, given hash functions h_1, h_2, h_3, we let $h_t^{(i)}(x) = h_t(x) \bmod r_i$, where $r_i = \mathcal{O}(|S_i|)$ is a power of two to be specified later. Since we choose ranges that are powers of 2, observe that for $r_i < r_j$ we have $h^{(i)}(x) = h^{(j)}(x) \bmod r_i$.

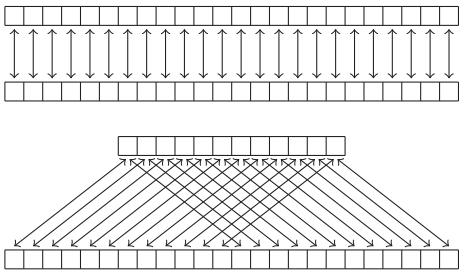


Figure 5.1: Computing the common elements in two batmaps is done using pairwise comparisons. For the sake of the illustration we have drawn each batmap as an array. For batmaps of the same size, we simply need to compare elements at the same position (top). For batmaps of different sizes, each entry in the smaller batmap needs to be compared to several entries in the larger batmap (bottom).

	$\underline{x}z$				$\underline{y}$
	$\underline{z}$			$\underline{x}y$	
$\underline{y}$		$x\underline{z}$			

Figure 5.2: Example of a 2-of-3 assignment for the set $S = \{x, y, z\}$. Each element has one possible position in each of the three hash tables. Two of these (where the element is underlined) are used to store the element. In the above example, an assignment of one element to each position exists, but there is always a probability (which we will bound in section 5.2.2) that no assignment exists.

This means that if $x \in S_j$ is stored in $(1, p_1)$ and $(2, p_2)$ it suffices to check positions $(1, p_1 \bmod r_i)$ and $(2, p_2 \bmod r_i)$ to determine if $x \in S_i$. Below, we explain how this principle can be used to efficiently count the number of items in $S_i \cap S_j$.

We will return to the issue of constructing the representation later, and for now simply assume that this is possible. Suppose that $x \in S_i \cap S_j$. Then, because we have stored x redundantly in the hash tables there exists at least one t for which $A_t^{(i)}[h_t^{(i)}(x)] = A_t^{(j)}[h_t^{(j)}(x)]$. For now we assume that $r_i = r_j = r$, which means that $h^{(i)} = h^{(j)}$ for all t. Now, by making all equality checks of the form "$A_t^{(i)}[p] == A_t^{(j)}[p]$", where $t \in \{1, 2, 3\}$ and $p \in \{0, \ldots, r-1\}$, we can identify each element in $S_i \cap S_j$. These comparisons, illustrated in Figure 5.1, parallelize very well. However, to count the number of elements in the intersection, an additional trick is needed. We can impose a cyclic order to the three hash tables, such that h_1 is followed by h_2, h_2 is followed by h_3, and h_3 is followed by h_1. Then for an occurrence of x in a hash table it makes sense to ask whether the other occurrence of x is in the hash table is before or after (it will be in exactly one of these). We use a single bit per position p in the hash tables to store this information, denoted $b_t^{(i)}[p]$. Consider a pair of items $\{i, j\}$, and a position (t, p) in their batmaps (assumed to be of the same size). In order to only count exactly once a transaction x where both items appears, we use the condition $(A_t^{(i)}[p] = A_t^{(j)}[p]) \wedge (b_t^{(i)}[p] \vee b_t^{(j)}[p])$ to determine if the elements in position p are overlapping and should be counted. See Figure 5.3 for an illustration. It is easy to check that in both the case where an element x is stored in the same two hash tables in both batmaps, and the case where there is only one overlapping occurrence, x is counted exactly once. We will see later that there will be positions p in each of $A_t^{(i)}$ that contain no element from S_i — in these positions we simply set $A_t^{(i)}[p] = \perp$ and $b_t^{(j)}[p] = 0$ to ensure that no counting is done. Here $\perp$ is a NULL value that is not in any set S_i.

For the general case, since r_i divides r_j, each position in $A^{(i)}$ corresponds to r_j/r_i positions in $A^{(j)}$ as explained above. That is, we can again count the number of elements in $S_i \cap S_j$ by comparing each position in $A^{(j)}$ with a position in $A^{(i)}$ (see Figure 5.1).

Compression Since our method is based on hashing we can use a compression scheme that stores each item relative to the set of items with the same hash value (see section 5.3.1 for details). This gives a significant space saving for dense sets: in our implementation each hash table entry uses just 8 bits, including $b_t^{(i)}[p]$, whenever the density of a set is above 2^{-8}.

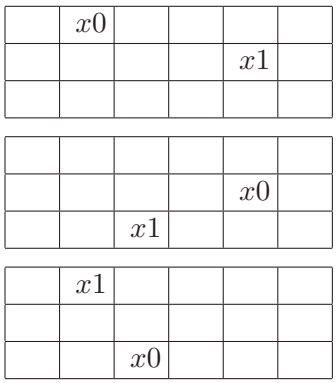


Figure 5.3: The three possible 2-of-3 assignments with respect to a single element x. Along with each occurrence is the bit that tells whether this occurence is before or after the other occurrence in the circular order of rows. When counting the common elements in two data structures we use this information to only count the last occurrence, in case the data structures store an item x in the same two positions. This is accomplished by a logical `OR` of the associated bits.

5.2.1 Data structure construction

We employ an insertion procedure that generalizes cuckoo hashing [76] (which places elements in 1 of 2 possible positions). The idea is to push elements around until an element is placed in a vacant position (with content $\perp$). An insertion of x starts by putting x in A_1, kicking out any element that might reside in $A_1[h_1(x)]$, making it *nestless*. In case there is a nestless key, it is inserted in A_2 in the same fashion, and so on using the circular order $1, 2, 3, 1, 2, 3, \ldots$. If the number of element moves exceeds a threshold `MaxLoop` the procedure returns the element that is currently nestless (our analysis below shows that this is a small probability event). The pseudo code is as follows (where $\leftrightarrow$ is used to denote the swapping of two variable values).

```
function INSERT(τ)
  loop MaxLoop times
    τ ↔ A1[h1(τ)]
    if τ = ⊥ then return ⊥
    τ ↔ A2[h2(τ)]
    if τ = ⊥ then return ⊥
    τ ↔ A3[h3(τ)]
    if τ = ⊥ then return ⊥
  end loop
  return τ
end
```

Since we need two occurrences of each element x, the insert procedure is called twice for each element. In case one of these insertions fails, we delete any occurrences of x and re-insert the nestless element returned (unless it happens to be identical to x). In the Analysis section below we bound the probability of insertions to fail. While this probability is low for a single set, failed insertions are likely to occur when handeling many sets. We describe how we handle failed insertions in Section 5.3.3.

5.2.2 Analysis

Suppose we have a data structure for a set S, with hash functions of range r. We now consider what might happen when we insert an element x_1 using the insert procedure. Possibly, a single copy of x_1 has already been inserted in the hash table. All other elements exist in exactly two copies. When moving an element it may happen that it is moved to the location of the other copy of that element. In this case the other copy is then moved to the third location, which must contain a different element. We consider the *transcript* of the insertion, which is the sequence of values of the variable τ from the INSERT() function after each element move upon insertion of x_1.

We first look at the possibility that each *copy* of an element appears only once in this sequence, i.e., that each element appears at most twice. Then each prefix of the transcript has the form $x_1^{d_1}, x_2^{d_2}, \ldots, x_k^{d_k}$, where $x_1, \ldots, x_k$ are distinct and $d_1, \ldots, d_k \in \{1, 2\}$ (number of copies that we move). Each such sequence appears with probability r^{1-k}, since we have a hash collision between x_i and x_{i+1} for $i = 1, \ldots, k-1$, and each such collision happens independently with probability at most $1/r$. Taking the union bound over all choices for $x_2, \ldots, x_k$ and $d_1, \ldots, d_k$ we get an upper bound on the probability that a transcript prefix of length k occurs:

$$2^k n^{k-1} r^{1-k} = 2\,(2n/r)^{k-1}.$$

The next case to consider is when the transcript involves the same copy of an item more than once (a *loop*). Then it is not hard to realize that the insert procedure will move a prefix of the elements in the transcript back to their original positions, and eventually have $\tau = x_1$ again. Then x_1 is pushed to a new table, and we again have two cases to consider.

1. The transcript does not again return to an element copy that appeared previously. Consider a prefix of the transcript of length k'. Then at least one of the two substrings of the transcript of length $k = \lfloor k'/3 \rfloor$ that start with x_1 will have no repeated element copies. We can bound the probability of such a transcript in the same way as above:

$$2^k n^{k-1} r^{1-k} = 2(2n/r)^{k-1} \leq 2(2n/r)^{k'/3-2}.$$

2. The transcript returns once again to a previously visited element copy (a second loop). Let k denote the number of distinct elements encountered. The number of transcripts starting with x_1 is then at most $2^k k^2 n^{k-1}$, where the k^2 factor is an upper bound on the number of ways the two loops can be formed. There are $k+1$ independent hash collisions for such a transcript, so each has probability r^{-k-1}, and by a union bound we see that this is an unlikely event when $r \geq (2+\varepsilon)n$:

$$2^k k^2 n^{k-1} r^{-k-1} = (2n/r)^k k^2/(nr).$$

Notice that the insertion may fail only in the last case. Using the assumption that $r \geq (2+\varepsilon)n$ we see that this happens for some k with probability at most

$$\begin{aligned}
&\sum_{k=1}^{n} (2n/r)^k k^2/(nr) \\
&\leq (nr)^{-1} \sum_{k=1}^{n} k^2 (1+\varepsilon/2)^{-k} \\
&= \mathcal{O}((\varepsilon^3 nr)^{-1}).
\end{aligned}$$

Here, we have bounded the sum by computing the integral wrt. k from 0 to ∞.

When the insertion succeeds, we see that the probability that it goes on for k' steps or more is bounded by $2(2n/r)^{k'/3-2}$. Thus, the expected number of steps is bounded by

$$\begin{aligned}
&\sum_{k'=1}^{\infty} 2(2n/r)^{k'/3-2} \\
&\leq \sum_{k'=1}^{\infty} (1+\varepsilon/2)^{-k'/3+2} \\
&= \mathcal{O}(1/\varepsilon).
\end{aligned}$$

Thus, by choosing $\varepsilon > 0$ as a constant, the expected time for performing all insertions is $\mathcal{O}(n)$.

5.3 Implementation

The implementation is split into two parts: code for execution at the GPU, and the pre- and postprocessing on the host system (CPU).

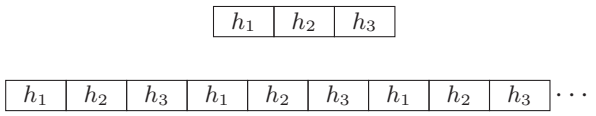


Figure 5.4: Organization of the three hash functions for B_0 (top) and B_i (bottom) where $|B_0| = 3r_0$. Each h_t above represents r_0 batmap elements covered by that hash function.

5.3.1 Layout of data structures

Our actual implementation differs a bit from the abstract description in Section 5.2. We compress the data so that only 8 bits are used per batmap element, while still being able to handle densities larger than 2^{-8}. Define three permutations, $\pi_t : \{1, \ldots, m\} \to \{1, \ldots, m\}$ for $t \in \{1, 2, 3\}$, let as earlier r_i denote the domain size of the hash functions for batmap B_i, and define the hash functions $h_t^{(i)}$ by

$$h_t^{(i)}(x) = |B_0| \left\lfloor \frac{\pi_t(x) \bmod r_i}{r_0} \right\rfloor + (\pi_t(x) \bmod r_0) + (t-1)r_0.$$

The batmap layout induced by these hash functions is illustrated in Figure 5.4. An important observation is now, that instead of storing element x at position $h_t^{(i)}(x)$ we could just as well store $\pi_t(x)$ at that position—the result of the element-wise comparisons between two batmaps would be the same. Next, by definition of $h_t^{(i)}$ the position of $\pi_t(x)$ (the stored representation of x) in a batmap uniquely identifies the least significat bits in $\pi_t(x)$, so explicitly storing these can be considered superfluous. Therefore, instead of storing x we will only store the 7 most significant bits of $\pi_t(x)$. That is, $\pi_t(x)$ can now be deduced from the position and the 7 bits stored in that position. Furthermore, we use 1 additional bit per batmap element to store the indicator bit $b_t^{(i)}[p]$ described in Section 5.2, and organize the bits so the indicator bit is the most significant of the 8 bits. This compression gives us 4 elements per 32-bit integer.

To get an idea of the efficiency of this compression scheme, assume that we have to shift s bits to the right in order to move the 7 most significant bits down to the least significant bits. Then $\log(m+1) - s \leq 7$, and consequently $2^s \geq (m+1)/128$. Also, as each element's position in a batmap should uniquely identify the least significant s bits of $h_t^{(i)}$ all hash domains must be at least of size $r_i \geq 2^s$ for this compression to work. If we compare to the uncompressed case with hash domain sizes of $2 \cdot 2^{\lceil \log(|S_i|) \rceil} \approx 2|S_i|$, we only obtain an actual compression (space reduction) when the input is sufficiently dense, i.e. where the set size is satisfying $2|S_i| \geq 2^s$, or equivalently $|S_i| \geq (m+1)/256$.

In the GPU, the actual comparisons are done in chunks of 32-bit integers (4 batmap elements at a time) in a way that completely avoids conditional statements: let x and y denote two 32-bit integers, and for convenience, let the 7 least significant bits in each 8-bit block be referred to as the *element bits* as they refer to a batmap element. If $\oplus$ denotes a logical `XOR` and "$(\cdots)_{16}$" means hexadecimal notation then

$$p = ((x \oplus y) \vee (\texttt{80808080})_{16}) - (\texttt{01010101})_{16}$$

gives a 0 (not 1) in the indicator bits iff the corresponding element bits of x and y are equal. To negate these bits, and only count a match if one of the corresponding indicator bits is set, define

$$p' = (p \oplus (\texttt{ffffffff})_{16}) \wedge ((x \vee y) \wedge (\texttt{80808080})_{16}).$$

We then account for $((p' \gg 7) + (p' \gg 15) + (p' \gg 23) + (p' \gg 31)) \wedge 7$ matches among the 2×4 elements represented by x and y. Here, $\gg$ denotes the shift operator as usual.

5.3.2 Our adaption of the GPU execution model

The execution model in GPUs and OpenCL can roughly be outlined as follows: a *kernel* is a set of instructions to be evalutated on a set of cores in a multiprocessor, and a thread running such a kernel is in OpenCL referred to as a *work item*. These work items can be organized in a one, two or three dimensional grid of size $W_1 \times W_2 \times W_3$, also referred to as a *work group*, and each running kernel instance can retrieve its coordinate (*local index*) in this grid. Also, we define the *global* data size as a multiplum of the work group size $G_1W_1 \times G_2W_2 \times G_3W_3$. When executing the kernel, work groups are generated by iterating over the global size, i.e. a total of $G_1G_2G_3$ work groups are formed. As with the local index, each kernel instance can retrieve its work groups' current global coordinate (*global index*) in this iteration process. As an example, consider a kernel that processes a two-dimensional 3200×3200 pixels image in chunks of 16×16 tiles. This would correspond to a work group size of 16×16 threads, a global data size of 3200×3200, and consequently $200 \cdot 200 = 4000$ work group positions in the global data.

OpenCL operates with multiple memory spaces, but here we will only refer to two of these: the most plentiful memory space, *global* memory, is the only memory space accessible from the host device (the CPU), and it has the largest latency among all the memory spaces. The low-latency *shared* memory resides closer to each compute unit, it is relatively small (e.g. around 16 kb), and is shared among all the threads in a work group. One of the most important considerations when implementing efficient algorithms for execution at GPUs is coalescing global memory accesses, and we achieve this by following best practice as described in [68]. In short, global memory

access by threads of a *half warp* (16 threads) are coalesced by the device in as few as one transaction when certain access requirements are met, e.g. if the 16 threads access a 64 bytes aligned segment, corresponding to 16 32-bit integers.

We adapt the GPU execution model to the ideas described in Section 5.2 and 5.3.1 in the following way: a list containing all n batmaps is transferred once to the device, and we then define the global size to be $n \times n$, and the work groups to be of size 16×16. Consequently, a total of n^2 batmap comparisons will be made, in chunks of size 16. The thread with local index (l_i, l_j) and global index (g_i, g_j) will now handle the comparisson of batmap $B_{16g_i+l_i}$ and $B_{16g_j+l_j}$ in turns of 16 integers (holding 64 batmap elements): each of the 256 threads in the work group first copies two single items from the input, which resides in global memory, into two small 16×16 integer arrays in shared memory. Each row in these small arrays correspond to a 16 integer wide slice of batmap B_{16g_i} to B_{16g_i+15}, and B_{16g_j} to B_{16g_j+15}, respectively. Because of coalescing, this copying is very efficient. Second, after synchronising the threads with a *memory barrier*, the 16-item wide batmap slices are now compared as described above, and the process is repeated with another copying from global to shared memory. This continues until all slices of the relevant batmaps have been compared.

5.3.3 Pre- and post processing

As the batmap comparisons are performed in the GPU in quantums of 2 times 16 consecutive batmaps the computation time of each such 16-block will be determined by the longest of these batmaps. Therefore, as a first step, we sort the batmaps by increasing width (corresponding to sorting the sets S_i by size), resulting in a strongly reduced computation time for the subresults for narrow batmaps. That is, after sorting we have $|B_i| \leq |B_j|$ for $i < j$.

Many graphics devices have a few-second hard limit on the execution time when the device is also used to support the display. Therefore, we break the GPU calculation into smaller parts of size $k \times k$ where k, in our experiments, typically had a value of 2048. Let $Z_{p,q}$ be a matrix holding the subresults for batmaps B_{pk} to B_{pk+k-1} and B_{qk} to B_{qk+k-1}. The division into smaller sub problems now has the convenient side effect that we, due to symmetry, only need to compute $Z_{p,q}$ for $p \leq q$, thereby cutting almost half of the GPU computation time, from n^2 to around $\binom{n}{2}$.

Failed insertions As there is a positive probability that some of the cuckoo insertions will fail due to collisions with previously inserted elements we need to handle these failed insertions separately. Let F_b be the set of items i for which insertion of value b in batmap B_i failed, and let A_b denote *all* items in input associated with b. For all transactions b, we construct

the pairs $(\min(a,c), \max(a,c))$ for which $a \in F_b$ and $c \in A_b$, and store each pair in a set $M_{p,q}$ where $(p,q) = (\lfloor \min(a,c)/k \rfloor, \lfloor \max(a,c)/k \rfloor)$. Whenever a subresult $Z_{p,q}$ is returned from GPU we extend it with the pairs found in $M_{p,q}$ before reporting the number of pairs found. (For $p = q$, only the upper triangle of $Z_{p,q}$ is reported because of symmetry.)

5.4 Experiments

Hardware setup All experiments were run on a MacPro with two Intel Xeon 5462, 2.8 GHz, 4-core CPUs and 6 GB RAM (bus speed 1.6 GHz), running Mac OS X 10.6. The machine had a GeForce GTX 285 graphics card with 1 GB RAM and 30 1.4 GHz cores having 8 computation units each. We observe that the two Xeon chips (combined) and the GPU have a similar complexity, with a total of 1.6 and 1.4 billion transistors, respectively[1]. However, the price of the 2 CPUs is significantly higher than that of the GPU (the factor is around 5 based on Intel's initial price for Xeon 5462, but this ratio has likely decreased somewhat). A specified indicator of the maximal energy consumption (TDP) is 2×80 W for the Xeon CPUs, and 204 W for the GPU, so the energy consumption at full utilization is likely to be similar.

5.4.1 Frequent pair mining

We have implemented the frequent pair mining with batmaps in Python, using the PyOpenCL interface to OpenCL, and compare our algorithm with Apriori [4, 15, 16] and FP-growth [17, 45]—both implemented by Christian Borgelt. Some experiments on Eclat [105] were also performed but it was significantly slower than the other three implementations and has therefore been left out of the graphs. Even though other implementations have been reported to be faster in some cases (e.g. [81, 93]), we found that the implementations available in the FIMI repository did not compile with recent versions of `gcc`. Thus, we have settled for Borgelt's implementations, that are generally regarded as state-of-the-art, as witnessed by a total of 35 citations in 2008-2010. Each test run had a hard limit of 1800 CPU seconds before it was cancelled.

The first set of experiments illustrate the behavior of the three algorithms when keeping the instance size constant and varying either the number of distinct items or the item density. An instance was generated by, for each transaction, including each of the n distinct items with probability p, and continue adding transactions until the desired total instance size was reached.

[1] Manufacturer's specification.

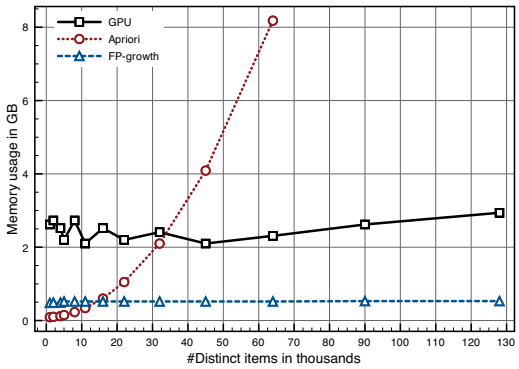


Figure 5.5: Memory usage for varying number of distinct items n, while holding the instance size at a constant 10 million items with an item density of 5%. Apriori scales poorly with n.

Figure 5.5 depicts the memory usage for the three algorithms for varying number of distinct items n. The space usage of the GPU implementation comes from the preprocessing, which is done on the CPU. We did not attempt to optimize the space usage of our preprocessing procedure, so it is likely that significant savings could be obtained by a space-aware implementation. From the plot we se that while both FP-growth and the GPU implementation scale well with n, Apriori has quadratic memory usage and exceeds the 6 GB RAM for less than 64,000 items.

Figure 5.6 compares the pure pair generation times for varying number of distinct items, but keeping the data size fixed. This is the part of all three methods that has super-linear complexity, so focusing on this allows us to see the asymptotic behavior more clearly. Not surprisingly is $n = 64,000$ an upper bound on what can be run with Apriori within the time limit, due to memory trashing. As expected, FP-growth exhibits linear growth in time usage as the number of items increases.

Figure 5.7 shows the total execution times including pre- and postprocessing. Our implementation suffers from high preprocessing times, partly due to our choice of Python (which is interpreted) as language. Still, our implementation outperforms Apriori and FP-growth for large n. According to a popular benchmark [11], Python executes between 2 and 106 times slower than GNU `C++` with a median of 49. We therefore believe that an optimized implementation of the preprocessing in `C` would achieve at least 1 order of magnitude speedup compared to our simple Python implementation.

We tested the behavior of the algorithms for varying item densities, and the results can be seen in Figure 5.8. While both Apriori and FP-growth have difficulties handling dense instances, our GPU implementation uses time almost independent of density. It can be noticed that for low densities the GPU time actually increases. This is due to the lower bound on space

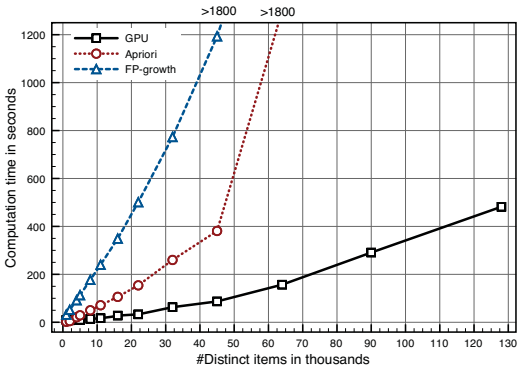


Figure 5.6: Computation times on pure pair generation for varying number of distinct items, while holding the instanse size at a constant 10 million items with an item density of 5%. Both Apriori and FP-growth exceeds their time limit on 1800 seconds when solving the $n = 64,000$ instance. In comparison, the GPU implementation scales well in n.

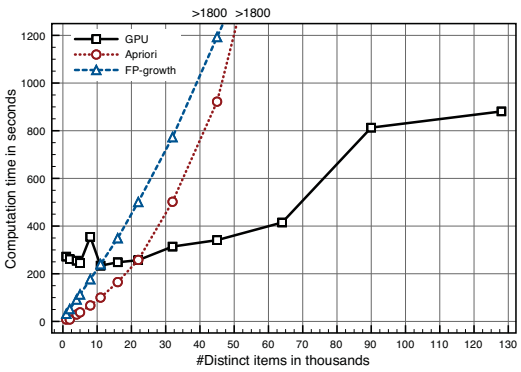


Figure 5.7: Total computation times, including pre- and postprocessing for varying number of distinct items, while holding the instance size at a constant 10 million items with an item density of 5%. The preprocessing time for the GPU implementation is high, but scales well in n.

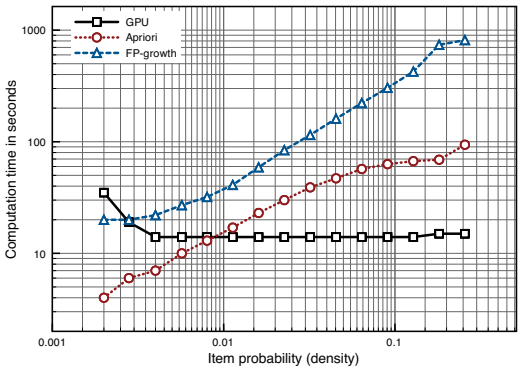


Figure 5.8: Computation times on pure pair generation for varying item density, while holding the instance size and number of distinct items constant at 10 million and 8000, respectively.

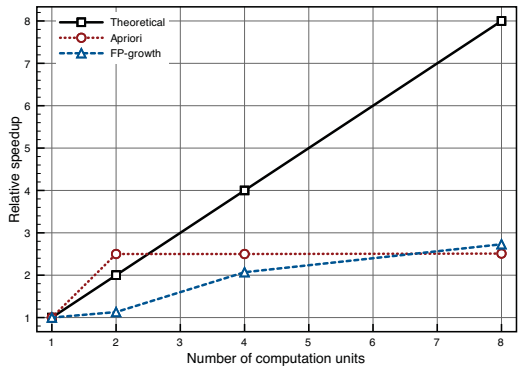


Figure 5.9: The relative speed-up vs. the number of computation cores. The theoretical speed-up is linear, but neither the implementation of Apriori nor FP-growth were benefitting noticably from more than four cores.

requirement for our compression scheme as described in Section 5.3.1.

In Figure 5.9 we try to illustrate how Apriori and FP-growth might scale to a larger number of computation cores. Our experiments was based on an instance of size 10 million items, 4000 distinct items, and a density of 5%. In a test simulating parallel execution on i cores, we split the original instance into i smaller instances of identical size. We compare the maximum execution times of test runs for $i \in \{1, 2, 4, 8\}$. As seen in the figure, none of the algorithms benefit noticeably from more than four cores. This is consistent with previous work which also finds that Apriori scales poorly on many processors [103].

The last experiment, seen in Figure 5.10, compares the algorithm performances on a "real-life" data set, WebDocs, which associates web documents and words. The data set was taken from the Frequent Itemset Mining

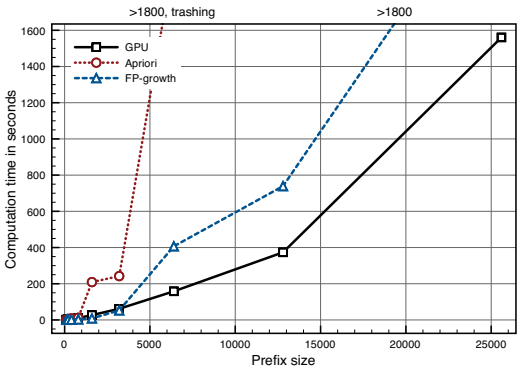


Figure 5.10: Computation time for pure pair generation for increasing prefix sizes of the WebDocs instance. The number of distinct items increases rapidly which explains why the computation time for Apriori explodes for small prefixes. None of the algorithms could solve a prefix of size 51,200 within the 1800 seconds time limit, and the memory usage of helper data structures for the GPU implementation exceeded the 6 GB RAM available.

Dataset Repository[2]. As WebDocs is an enormous instance we run several tests on prefixes of varying size. The number of distinct items in this instance increases rapidly so all three algorithms are challenged. As seen, Apriori exceeds the time limit first due to memory trashing. The GPU algorithm solves the largest instance: a 25.600 line prefix.

Processing speed computation The number of items processed by the GPU for a pair mining run can be estimated as follows. Consider the experiment with $n = 4000$ distinct items, a total instance size of 10^7, and $p = 5\%$. Sets in this instance have average size $10^7/4000 = 2500$, which means that each batmap is $3 \cdot 2^{\lceil \log(2 \cdot 2500) \rceil} = 3 \cdot 2^{13}$ bytes wide. Thus the combined input size to all set intersections is $4000^2 \cdot 3 \cdot 2^{13}$ bytes. The experiment used 10.87 seconds on the GPU and thus we processed 36.2 Gbyte per second. The memory bandwidth on the GPU, however, is around 159 Gbyte per second so we are a factor of over 4 from the theoretical maximum memory throughput.

To get a number that can be compared directly with CPU implementations based on merging, we compute the number of set elements processed per second. The total input size (in terms of number of set elements) to all set intersections is $4000^2 \cdot 2500 = 40 \cdot 10^9$. Thus, we processed $3.68 \cdot 10^9$ elements per second, which is typical for intersections of this size. Due to rounding of the size of hash tables, batmaps of the same size would be able to accommodate up to 63% more elements, which would give a maximal

[2] `http://fimi.cs.helsinki.fi/data/`

processing speed of $6 \cdot 10^9$ elements per second. On the other hand, if the rounding works against us, the processing speed would be only half of this.

5.4.2 Comparison with merging

A widely used representation of sets, that allows efficient computation of intersections, is sorted lists. A simple for-loop can be used to report all common elements, by scanning both lists. Even though this algorithm is extremely simple, it runs slowly on modern CPUs due to branch mispredictions. We performed an experiment in which we counted the number of identical elements in two sorted arrays of 2^{24} integers (32 bits each), repeated 100 times. The implementation was written in `C`, and compiled with `gcc` with optimization level `O3`. Doing one such run took 14.89 seconds (on one core), which means that $2.25 \cdot 10^8$ elements are handled per second. This is 13–26 times slower than the processing speed on the GPU.

To compare against a parallel implementation, we did 8 simultaneous runs (using 8 cores), which took 15.66 seconds. Since the time did not grow noticeably, we conclude that the computation does not (yet) have a memory bottleneck. The number of set elements processed per second using 8 cores is $1.71 \cdot 10^9$, or 29–57% of the throughput of the GPU batmap computed above. This means that performance is noticeably poorer on the CPUs than on the far less expensive GPU.

5.5 Conclusion

We have shown that a GPU allows set intersection and frequent pair mining that extends to much larger number of items than previous algorithms. Further, we believe that our approach may be pushed further with careful tuning, as we are still an order of magnitude from using the full memory bandwidth of the GPU. Our techniques may open up for new applications of e.g. association mining where there are tens of thousands of variables (e.g. genetic data).

One problem we leave open is to achieve similar results for intersections of more than two sets. There are two ways in which our work could possibly be extended: one is to use a generalization of batmaps that store items in d out of $d+1$ places. This would ensure that itemsets of size up to d would have at least one position witnessing their intersection. Another is to use batmaps to count, for each item in S_{i_1}, how many times this item appears in $S_{i_2}, S_{i_3}, \ldots$. At the end one would need to sum up the counts for the two occurrences of each item to determing if the item appeared in all sets.

Acknowledgements. We would like to thank Anna Pagh for taking part in showing theoretical results on our generalization of the cuckoo hashing insertion procedure, and Kumar Lav for his participation in initial exper-

iments with the method described in this chapter. This work was supported by a grant from the Danish National Research Foundation, as part of the project “Scalable Query Evaluation in Relational Database Systems”.

Chapter 6

Perspectives

In the previous chapters we considered the problem of joining exactly two relations, R_1, R_2, or equivalently, computing products $M_1 M_2$ of exactly two matrices. In this chapter we put this into perspective by considering how to handle products of more than two matrices and how to reduce matrix multiplication to combinatorial graph problems.

6.1 Triangles

Section 4.3 on page 34 described how database relations R_i can be represented by sparse Boolean affinity matrices M_i. It also described how the join of two relations $R_1 \bowtie R_2$ and multiplication of two matrices $M_1 M_2$ can be modelled in sequentially tripartite graphs $G = (V, E)$ with $V = V_1 \cup V_2 \cup V_3$ and $E = (V_1 \times V_2) \cup (V_2 \times V_3)$. Recall that, in this model, an output pair of $R_1 \bowtie R_2$ or a non-zero entry in $M_1 M_2$ are equivalent to a path of length 2 from V_1 to V_3.

Assume that $|V_1| = |V_2| = |V_3| = n$. We can naively reduce Boolean matrix multiplication to triangle reporting by introducing n^2 new edges $E' = \{(i, j) \mid i \in V_1, j \in V_3\}$ and noticing that the subset of $V_1 \times V_3$ that is part of the output consists exactly of the vertices that are part of a triangle in $G' = (V, E \cup E')$.

Triangle reporting, and reductions to and from matrix multiplication have been studied intensively: By using matrix multiplication Alon et al. [6] counted triangles in $\mathcal{O}(N^{2\omega/(\omega+1)}) \leq \mathcal{O}(N^{1.41})$ time, for $\omega \leq 2.376$, thereby reducing Boolean matrix multiplication to triangle reporting. This reduction was done using a similar technique as the one described in Chapter 4 where nodes are split according to a degree threshold and high-degree nodes are fed as input to a Boolean matrix multiplication algorithm whereas low-degree nodes are handled by an enumerating algorithm. Schank and Wagner [86] gave a practical survey of multiple triangle finding algorithm implementations. Williams and Williams [98] presented a combinatorial reduction from

fast triangle detection in general graphs to Boolean matrix multiplication and furthermore reduced it to a number of related problems. The reduction did not exploit sparsity of the graphs. Lately, in 2010, Kolountzakis et al. [56] presented a new algorithm for counting the number of triangles in a graph.

6.1.1 A probabilistic reduction

In the following we describe a non-trivial combinatorial probabilistic reduction by Pagh [75] from triangle reporting to Boolean matrix multiplication. The reduction exploits sparsity of the graphs. Consider two $n \times n$ Boolean matrices M_1 and M_2, and let as earlier N denote the total number of non-zeroes in M_1 and M_2, and Z the number of non-zeroes in the product $M_1 M_2$. The reduction utilizes the following three problems:

- TriangleReporting(n, N, Z): Report the set of all the, at most Z, triangles that can be found in an undirected graph with n vertices and N edges.
- TriangleEdges(n, N): Report the set of edges that are part of a triangle in a graph with n nodes and N edges.
- BooleanMatrixMultiplication(n, N, Z): Compute the Boolean matrix product of two $n \times n$ matrices with a total of N non-zeros and with at most Z non-zeros in the product.

It makes sense to describe the complexity of TriangleReporting as the sum of a cost $f(n, N)$ that is unrelated to the output and a cost $f'(n, N)$ per triangle reported.

Lemma 6.1 *If* TriangleReporting(n, N, Z) *can be solved in* $\mathcal{O}(f(n, N) + Zf'(n, N))$ *time, where* f *and* f' *are increasing functions, then* TriangleEdges(n, N) *can be solved in*

$$\mathcal{O}((f(n, N) + Nf'(n, N)) \log n)$$

time with probability $1 - \mathcal{O}(n^{-1})$*, and the output is always correct.*

Proof We find all triangle edges by considering V_1, V_2 and V_3 in turn and, for each nodeset, generating the edges in $\log n$ iterations.

First, consider the turn with focus on V_1. Randomly order the vertices in V_1 and consider the subgraph where only the first 2^t nodes in V_1 and their incident edges are present. That is, when calling TriangleReporting, only triangles involving the first 2^t nodes from V_1 are reported. For each of these, we report the relevant triangle edges in $V_2 \times V_3$, and whenever an edge has been reported, we remove it from the graph in the next iterations. All triangle edges in $V_2 \times V_3$ are reported no later than at the last iteration.

A similar process is applied to report edges in $V_1 \times V_2$ and $V_1 \times V_3$, so with $\mathcal{O}(\log n)$ calls to TriangleReporting we immediately get the first term, $\mathcal{O}(f(n, N) \log n)$, in the lemma.

To account for the per-edge cost, consider again the first turn focusing on V_1, an arbitrary edge $e \in V_2 \times V_3$ and the set $B \subseteq V_1$ of nodes that are part of a triangle containing e. If $|B| < \log n$, then clearly at most $\mathcal{O}(\log n)$ triangles containing e will be reported throughout all $\log n$ iterations.

If $|B| \geq \log n$ we argue, that with probability $1 - \mathcal{O}(n^{-3})$, we will still only see at most $\log n$ triangles containing e throughout the $\log n$ iterations. After the first iteration t in which any node from B is seen in the 2^t-sample, no more triangles will be found in subsequent iterations, since e is removed. We can bound the probability that 0 triangles are found within the first $t - 1$ iterations and more than $\log n$ triangles are found exactly at iteration t: Let X_v be $|V_1|$ Boolean random variables with $X_v = 1$ iff $v \in V_1$ is among the 2^t sampled vertices, and let $X = \sum_{v \in B} X_v$. Note that X_v are negatively dependent, and that Chernoff bounds still holds for this case [32]. For $0 < c_1 < c_2$ we use a Chernoff bound for sampling without replacement and appropriate assignments of c_1 and c_2 to bound the following cases:

1. A node from B is found with high probability whenever the sample size 2^t is sufficiently large:

$$\frac{|B|}{|V_1|} 2^t > c_1 \log n \Rightarrow \mathbf{Pr}[X \geq 1] \geq 1 - n^{-3}$$

2. For sufficiently small samples sizes 2^t, the probability that we sample a lot of nodes from B is small:

$$\frac{|B|}{|V_1|} 2^t \leq c_2 \log n \Rightarrow \mathbf{Pr}[X \geq c_2 \log n] \geq n^{-3}$$

As each edge with probability $1 - \mathcal{O}(n^{-3})$ is reported at most $\mathcal{O}(\log n)$ times throughout all $\log n$ iterations we obtain the last term of the lemma: $\mathcal{O}(N f'(n, N) \log n)$. The probability of $1 - \mathcal{O}(n^{-1})$ is obtained by taking the union bound of all $N \leq n^2$ edges. □

Lemma 6.2 *If* TriangleEdges(n, N) *can be solved in* $f''(n, N)$ *time, then* BooleanMatrixMultiplication(n, N, Z) *can be solved in*

$$\mathcal{O}(f''(n, N + 4Z) \log n)$$

time.

Proof The main idea is to *collapse* the node sets V_1 and V_3 into increasingly bigger sets of size 2^t for $t = 0, 1, \ldots, \log n$. By collapsing a set of nodes,

we, roughly speaking "replace" the nodes with a single representative, and let all the involved edges be incident to that representative instead. This is formalized below. In each iteration we compute sets of edges between the collapsed nodes until we reach the result no later than at iteration $t = \log n$.

Let $I_{t,u} = \{i_u \in V_1 \mid \lceil 2^t p/n \rceil = u, p \in [n]\}$ and $J_{t,v} = \{j_v \in V_3 \mid \lceil 2^t p/n \rceil = v, p \in [n]\}$ so that $I_{t,u} \subseteq V_1$ and $J_{t,v} \subseteq V_3$ are decreasingly smaller sets as t grows. In iteration t we compute the set

$$R_t = \{(u, v) \mid \text{there is a path from } I_{t,u} \text{ to } J_{t,v}\}.$$

In each iteration t we create the graph of collapsed nodes induced by $I_{t,u}$ and $J_{y,v}$, i.e. with the vertex set $V_t = \{i_u, k_w, j_v \mid u, v \in [2^t], w \in [n]\}$ and edge set

$$E_t = \{(i_u, k_w) \mid E \cap (I_{t,u} \times V_2) \neq \emptyset, u \in [2^t], w \in [n]\} \cup \{(k_w, j_v) \mid E \cap (V_2 \times J_{t,v}) \neq \emptyset, w \in [n], v \in [2^t]\}.$$

Note that, if the edges (i_u, k_w) or (k_w, j_v) are in E_t, then the edges $(i_{\lceil u/2 \rceil}, k_w)$ or $(k_w, j_{\lceil v/2 \rceil})$ must be in E_{t-1}, respectively, and consequently

$$R_t \subseteq \{(u, v) \mid (\lceil u/2 \rceil, \lceil v/2 \rceil) \in R_{t-1}\}.$$

R_{t-1} thus identifies a set T_t of at most $4Z$ edges that may complete a triangle if added to E_t. The graph $G_t = (V_t, E_t \cup T_t)$ has at most $N + 4Z$ edges, and running TriangleEdges on G_t would identify the edges of R_t. In no later than iteration $\log n$ we have the result. □

Corollary 6.3 *If* TriangleReporting(n, N, Z) *can be solved in*

$$\mathcal{O}(f(n, N) + Zf'(n, N))$$

time, then, with probability $1 - \mathcal{O}(n^{-1})$, BooleanMatrixMultiplication(n, N, Z) *can be solved in*

$$\mathcal{O}((f(n, N + 4Z) + (N + 4Z)f'(n, N + 4z)) \log^2 n)$$

time.

We have now seen reductions between triangle reporting and Boolean matrix multiplication. Due to the above reduction, a faster output sensitive algorithm for triangle reporting will imply a faster output sensitive algorithm for Boolean matrix multiplication. However, the oppositely directed reduction by Alon et al. [6] is not output sensitive, so a faster output sensitive algorithm for Boolean matrix multiplication does not immediately imply a faster output sensitive algorithm for triangle reporting.

We have now seen that join-projects, boolean matrix multiplication, and triangle reporting are related problems. Matrix multiplication and tringle reporting are both well studied problems, and as no known algorithms for these problems scale well in k, Hypothesis 1 from Chapter 1 is unlikely to hold.

6.2 Chain joins with projection

We can consider join-projects of $k \geq 3$ tables $R_i = (a_i, a_{i+1})$, and write them in relational algebra as $\pi_{a_1,a_{k+1}}(\bowtie_i R_i)$. Equivalently, we can write them as a product of multiple Boolean matrices $M = M_1 M_2 \cdots M_k$. As for the case with two operands, a join of $k \geq 3$ tables can be represented by a layered graph with $k+1$ node sets $V_1, V_2, \cdots, V_{k+1}$, in which V_i for $2 \leq i \leq k$ holdes the values of the join attribute of R_{i-1} and R_i, and k edge sets $E_1, \ldots, E_k$ where $G_i = (V_i \cup V_{i+1}, E_i)$ models the affinity matrix for R_i. Similar to the triangle case described above, it is not hard to see that by introducing $\mathcal{O}(n^2)$ new edges from V_1 to V_{k+1}, the problem of reporting output tuples of the join-project and the Boolean matrix product becomes equivalent to the problem of finding cycles of length $k+1$ in $G = (\cup V_i, (\cup E_j) \cup (V_1 \times V_{k+1}))$.

Yuster and Zwick [104] presented a generalization of their fast sparse matrix multiplication algorithm for the case with three or more matrices. The algorithm enumerates all paths in the layered graph described above and computes the tail product $M_r \cdots M_k$ using recursive calls to the algorithm. At the end of the recursion, their fast sparse multiplication algorithm is used. They compare to dense multiplication using $\mathcal{O}(n^\omega)$ time and the naive algorithm (equivalent to a classical merge-join) using at most $\mathcal{O}(nN)$ time. For $k = 3$ and $\omega \geq 2.376$ they obtain an improvement over both compared alternatives when $n^{1.24} \leq N \leq n^{1.45}$, and for $k \geq 4$ they are never the fastest.

If the final output is generated by an iterative process of multiplying two consecutive matrices, it is, in the general case, non-trivial to find an optimal pairwise association of the matrices. Consider the product of k matrices $M_1, \ldots, M_k$. If the matrices are dense, the optimal order of multiplication is solely dependent of the matrix dimensions and can be found efficiently using dynamic programming in $\mathcal{O}(k^3)$ time [41, 88]. Here, it is assumed that multiplying a $p \times q$ and a $q \times r$ (dense) matrix uses $\mathcal{O}(pqr)$ operations. Hu and Shin [48, 49] used $\mathcal{O}(k \log k)$ time to find an optimal order for dense matrices and transform the problem into the problem of partitioning a convex polygon into nonintersecting triangles. Chin [20] used $\mathcal{O}(k)$ time to calculate a near-optimal order which is guaranteed to be within a factor 1.25 of the optimal order.

For sparse matrices, the optimal order of multiplication may vary significantly, even for matrices of the same dimensions, and finding an optimal order therefore becomes increasingly complex. Cohen [23] presented a method to predict the non-zero structure of a product of two or more matrices, and this can be used to exploit the sparsity of intermediate products. She used a *reachability-set* size estimation algorithm [22] for arbitrary directed graphs to estimate the number of ancestors or descendents from a subset of designated nodes in the layered graph mentioned above. For sub-products $M_i \ldots M_j$, $i < j$, the reachability for the nodes in V_{i-1} and

V_{j+1} is estimated (and referred to as the *column sizes* and *row sizes*), and dynamic programming is then used to compute an optimal association with respect to these estimated row and column sizes of the sub-products. If a full reducer [80] is assumed applied to the join before estimation, Cohen's technique applies immediately to arbitrary acyclic joins with exactly two projections, e.g. chain-joins of the form $\pi_{a_p,a_q}(\bowtie_i R_i)$ where $R_i = (a_i, a_{i+1})$ and $1 \leq p < q \leq k+1$. Note the two generalizations of this, compared to the join-projects generally addressed in this text: firstly, the joins need not to be chain-joins as long as the join graph is acyclic; and secondly, there is no assumption of the projected columns to be located in the "end point"-relations of the join graph, for example at relation R_1 and R_k in the chain-join $\bowtie R_i$.

Also for arbitrary acyclic joins with exactly three projections in a chain (for example the chain-join $\pi_{a_p,a_q,a_r}(\bowtie_i R_i)$ for $1 \leq p < q < r \leq k$) the technique may also apply with a two-phase approach: for every value of a_q, count and multiply the number of paths from a_q to a_p and from a_q to a_r, and summarize these sub-results. Note that the projected attributes must be descendants for this to work, i.e. there needs to be a simple path from a_q to a_r. Therefore, join-graphs like a star-join, are generally not valid input.

For $k = 3$, our size estimation algorithm from Chapter 3 can be used to find the optimal order by efficiently estimating the sizes of M_1M_2 and M_2M_3, respectively: the pair chosen for the first multiplication uniquely identifies the remaining matrix for the second multiplication.

However, for $k \geq 4$, our estimation algorithm cannot be used directly. Having identified the input matrices for the first multiplication (M_1M_2, M_2M_3 or M_3M_4) does not immediately imply a preferable input for the second multiplication, simply because two sparse operands in matrix multiplication may have a dense product (consider the product M_1M_2, where the first column in M_1 and the first row in M_2 are 1, and all other entries are zero). Thus, the intermediate estimates must contain more information than just a number for the density. Using the size estimation described in Chapter 3, the algorithm depicted in Algorithm 3 takes an iterative greedy approach to multiplying k matrices using $\mathcal{O}(k)$ size estimations.

Algorithm 3 Computing the product $M_1M_2 \cdots M_k$ using $k-1$ estimations to initialize the algorithm, and at most 2 estimations for each of the $k-1$ iterations. The estimation technique described in Chapter 3 can be used.

1: Perform a size estimation of each of the $k-1$ consecutive matrix pairs
2: **for** $i \leftarrow 1, k-1$ **do**
3: Assume M_jM_{j+1} is the estimated most sparse product
4: Perform the multiplication $M' = M_jM_{j+1}$
5: In the following iterations, consider M' instead of M_jM_{j+1}
6: Perform size estimations of $M_{j-1}M'$ and $M'M_{j+2}$
7: **end for**

Part II

Vertical partitioning

Chapter 7

Vertical partitioning

In conventional row store database systems, table rows are physically organized in *pages*. Rows are accessed through a buffer manager that keeps a subset of the pages in RAM for faster access. Pages are typically also the unit of disk access, meaning that each read or write of a single row implies a read or write of the corresponding page.

Let us try to estimate the number of I/Os in a simplified situation with a single query on a single table with N rows. If a page can hold p rows and the buffer has room for B pages, then, for $N > Bp$, a table scan will cost at least around $N/p - B$ I/Os, and the I/O cost asymptotically grows linearly with $1/p$. The probability of any given page to exist in the buffer can be estimated to Bp/N if all slots in the buffer hold pages for our single table, so the probability of any given page *not* to exist in the buffer is likely to be larger than $1 - Bp/N$. Therefore, if the query accesses n random rows we can estimate the number of I/Os to be in the order of $n(1 - Bp/N)/p = n(1/p - B/N)$. Equivalently, the I/O costs of accessing a table seems to be inverse proportional to the row *width* of the table and proportional to the number n of accessed rows.

We should note the existence of Oracle's *Exadata* [70, 71], which is a combination of hardware and software that aims to speed up performance of disc access. The system price is in the order of $1,000,000 USD [72].

The basic idea in *vertical partitioning* is to minimize the amount of costly I/Os by partitioning tables vertically so that the number of irrelevant columns for a given query, and thus the relative row width $1/p$, is minimized. Execution of a query on a vertically partitioned table may, however, be more complex than executing the same query on an unpartitioned table: updates for a column must be performed at all partitions holding a replica of that column, and if referred columns are stored in multiple partitions these partitions must be joined. Reconstructing a row by a join of two or more tables may be more expensive than accessing the unpartitioned row, depending on the total width of the partition rows (including meta data),

search time, and the current contents of the buffer. Section 7.2.4 on page 80 elaborates the complexity of solving the vertical partitioning optimization problem.

The cost model described in this chapter has its origin in VoltDB, earlier named *H-store* [91]: a re-thinking of row store database systems from scratch, and built to scale well in the number of nodes in the database cluster. Therefore, the model presented here allows partitions to exist in different physical nodes. In VoltDB, rows are not organized in tables but stored individually in RAM only. This eliminates the concern with expensive I/O, but the main idea of reducing row widths is still valid in general due to spatial locality.

This chapter is an elaboration of the paper *Vertical partitioning of relational OLTP databases using integer programming* [7] that was presented at the 5th International Workshop on Self Managing Database Systems (SMDB 2010), Long Beach, California, USA.

7.1 Introduction

In this chapter we consider OLTP databases with an H-store [91] like architecture in which we would aim for maximizing the number of single-sited transactions (i.e. transactions that can be run to completion on a single site). Given a database schema and a workload we would like to reduce the cost of evaluating the workload. In row-stores, where each row is stored as a contiguous segment and access is done in quantums of whole rows, a significant amount of superfluous columns/attributes (we will use the term *attribute* in the following) are likely to be accessed during evaluation of a workload. It is easy to see that this superfluous data access may have a negative impact on performance so in an optimal world the amount of data accessed by each query should be minimized. One approach to this is to perform a *vertical partitioning* of the tables in the schema. A vertical partitioning is a, possibly non-disjoint, distribution of attributes and transactions onto multiple physical or logical sites. (Notice that vertical and horizontal partitioning are not mutually exclusive and can perfectly be used together). The optimality of a vertical partitioning depends on the context: OLAP applications with lots of many-row aggregates will likely benefit from parallelizing the transactions on multiple sites and exchanging small sub-results between the sites after the aggregations. OLTP applications on the other hand, with many short-lived transactions, no many-row aggregates and with few or no few-row aggregates would likely benefit from gathering all attributes read by a query locally on the same site: inter-site transfers and the synchronization mechanisms needed for non-single-sited or parallel queries (e.g. undo and redo logs) are assumed to be bottlenecks in situations with short transaction durations. Stonebraker et al. [91] and Kallman et al. [54] discuss the benefits

of single-sitedness in high-throughput OLTP databases in more details.

This chapter presents a cost model together with two algorithms that find either optimal or close-to-optimal vertical partitionings with respect to the cost model. The two algorithms are based on quadratic programming and simulated annealing, respectively. For a given partitioning and a workload, the cost model estimates the number of bytes read/written by access methods in the storage layer and the amount of data transfer between sites. Our model is made with a specific setting in mind, captured by five headlines:

OLTP The database is a transaction processing system with many short lived transactions.

Aggregates No many-row aggregates and few (or no) aggregates on small row-subsets.

Preserve single-sitedness We should try to avoid breaking single-sitedness as a large number of single-sited transactions will reduce the need for inter-site transfers and completely eliminate the need for undo and redo logs for these queries if the partitioning is performed on an H-store like DMBS [91].

Workload known Transactions used in the workload together with some run-time statistics are assumed to be known when applying the algorithms.

Furthermore, following the consensus in the related work (see Section 7.1.3) we simplify the model by not considering time spent on network latency (if all vertical partitions are placed locally on a single site, then time spend on network latency is trivially zero anyway). A description of how to include latency in the model at the expense of increased complexity can be found in Section 7.4 on page 81.

7.1.1 Outline of approach

The basic idea is as follows. We are given an input in form of a schema together with a workload in which queries are grouped into transactions, and each query is described by a set of statistical properties.

For each query q in the workload and for each table r accessed by q the input provides the average number n_r of rows from table r that is retrieved from or written to storage by query q. Together with the (average) width w_a of each attribute a from table r this generally gives a good estimate for how much attribute a costs in retrievals/writes by access methods for each evaluation of query q, namely $W'_{a,q} = w_a \cdot n_r$.

Given a set of sites, the challenge is now to find a non-disjoint distribution of all attributes, and a disjoint distribution of transactions to these sites so

that the costs of retrievals, writes and inter-site transfers, each defined in terms of $W'_{a,q}$ as explained in details below, is minimized. This means, that the primary executing site of any given query is assumed to be the site that hosts the transaction holding that query.

As mentioned above, our algorithms will not break single-sitedness for read queries and therefore no additional costs are added to the execution of read queries by applying this algorithm. In contrast, since the storage costs (the sum of retrieval, write and inter-site transfer costs) for a query is minimized and each tuple become as narrow as possible, the total costs of evaluating the queries (e.g. processing joins, handling intermediate subresults, etc.) are assumed to be, if not minimized, then reduced too.

7.1.2 Contributions

This chapter contributes with the following:

- an algorithm optimized for H-store like architectures, preserving single-sitedness for read queries and in which load balance among sites versus minimization of total costs can be prioritized arbitrarily,
- a more scalable heuristic, and
- a micro benchmark of a) both algorithms based on TPC-C and a set of random instances, b) a comparison between the benefits of local versus remote partition location, and c) a comparison between disjoint and non-disjoint partitioning.

7.1.3 Related work

A lot of work has been done on data allocation and vertical partitioning but to the best of our knowledge, no work solves the exact same problem as us: distributing both transactions and attributes to a set of sites, allowing attribute replication, preserving single-sitedness for read queries and prioritizing load balancing vs. total cost minimization. We therefore order the references below by increasing estimated problem similarity and do not mention work dedicated on vertical partitioning of OLAP databases.

In 1976 Eisner [33] reduced the cost of information retrieval by vertically partitioning records into a primary and a secondary record segment. This was done by constructing a bi-partite graph with two node sets: one set with a node for each attribute and one set with a node for each transaction. By connecting attribute and transaction nodes with a weighted edge according to their affinity, a min-cut algorithm could be applied to construct the partitioning.

Sacca and Wiederhold [83] assumed a set of horizontal and vertical fragments of a database was known in advance and produced a disjoint distribution of these fragments onto a set of network-connected processors using

a greedy first-fit bin packing heuristic. Similarly, Menon [63] distributed a set of predefined fragments to a set of sites, but used a linearized quadratic program to compute the solution.

Sarathy et al. [85] took as input a geographically distributed database together with statistics for a query pattern on this database and produced as output a non-disjoint distribution of whole database tables to the physical sites so that the total amount of transfer was minimized. They modelled the problem as a linearized quadratic program which was solved in practice using heuristics. The costs of joins were minimized by first transferring join keys and then transferring the relevant attributes for the relevant rows to a single collector site.

Navathe and Ra [67] constructed a disjoint partitioning with non-remote partition placement. They used an attribute affinity matrix to represent a complete weighted graph and generated partitions by finding a linearly connected spanning tree and considering a cycle as a fragment.

Cornell and Yu [28] generated a non-remote, disjoint partitioning minimizing the amount of disk access by recursively applying a binary partitioning. The partitioning decisions were based on an integer program and with strong assumptions on a System-R like architecture when estimating the amount of disk access.

Agrawal et al. [5] also constructed a disjoint partitioning with non-remote partition placement. They used a two-phase strategy where the first phase generated all relevant attribute groups using association rules [2] considering only one query at a time, and the second phase merged the attribute groups that were useful across queries.

Son and Kim [89] presented an algorithm for generating disjoint partitioning by either minimizing costs or by ensuring that exactly k vertical fragments were produced. Inter-site transfer costs were not considered. The partitioning was produced using a bottom-up strategy, iteratively merging two selected partitions with the best "merge profit" until only one large super-partition existed. The k-way partitioning was found at the iteration having exactly k partitions and the lowest-cost partitioning was found at the iteration with the lowest cost.

Chu and Ieong [21] minimized the amount of disk access by constructing a non-remote and non-disjoint vertical partitioning. Two binary partitioning algorithms based on the branch-and-bound method were presented with varying complexity and accuracy. The partitionings were formed by recursively applying the binary partitioning algorithms on the set of "reasonable cuts".

Chakravarthy et al. [18] did not present an algorithm but gave an interesting objective function for evaluating vertical partitionings. The function was based on the square-error criterion as given in [51] for data clustering, but did not cover placement of transactions which, in our case, has a large influence on the expected costs.

Navathe et al. [66] considered the vertical partitioning problem for three different environments: a) single site with one memory level, b) single site with several memory levels, and c) multiple sites. The partitions could be both disjoint and non-disjoint. A clustering algorithm grouped attributes with high affinity by using an attribute affinity matrix together with a bond energy algorithm [53]. Three basic algorithms for generating partitions were presented which, depending on the desired environment, used different prioritization of four access and transfer cost classes.

7.1.4 Outline of chapter

In section 7.2 we derive a cost model together with a quadratic program defining the first algorithm. Section 7.3 describes a heuristic based on the cost model found in Section 7.2, and Section 7.4 discusses a couple of ideas for improvements. Computational results are shown in Section 7.5.

7.2 A linearized QP approach

In this section we develop our base model, a quadratic program (QP), which will later be extended to handle load balancing and then linearized in order to solve it using a conventional mixed integer program (MIP) solver.

7.2.1 The base model

In a vertical partitioning for a schema and a workload we would like to minimize the sum

$$A + pB \tag{7.1}$$

where A is the amount of data accessed locally in the storage layer, B is the amount of data needed to be transferred over the network during query updates and p is a penalty factor.

We assume that each transaction has a primary executing site. For each transaction $t \in \mathcal{T}$, each table attribute $a \in \mathcal{A}$, and each site $s \in \mathcal{S}$ consider two decision variables $x_{t,s} \in \{0,1\}$ and $y_{a,s} \in \{0,1\}$ indicating if transaction t is executed on site s and if attribute a is located on site s, respectively. All transactions must be located at exactly one site (their primary executing site), that is

$$\sum_{t \in \mathcal{T}} x_{t,s} = 1 \quad , \forall s \in S \tag{7.2}$$

and all attributes must be located at at least one site, that is

$$\sum_{a \in \mathcal{A}} y_{a,s} \geq 1 \quad , \forall s \in S.$$

To determine the size of A and B from equation (7.1) introduce five new static binary constants describing the database schema:

- $\alpha_{a,q}$ indicates if attribute a itself is accessed by query q
- $\beta_{a,q}$ indicates if attribute a is part of a table that q accesses
- $\gamma_{q,t}$ indicates if query q is used in transaction t
- δ_q indicates if query q is a write query
- $\varphi_{a,t}$ indicates if any query in transaction t reads attribute a

Single-sitedness should be maintained for reads. That is, if a read query in transaction t accesses attribute a then a and t must be co-located:

$$x_{t,s}\varphi_{a,t} = 1 \Rightarrow y_{a,s} = 1 \quad , \forall t \in \mathcal{T}, a \in \mathcal{A}$$

or equivalently

$$y_{a,s} - x_{t,s}\varphi_{a,t} \geq 0 \quad , \forall t \in \mathcal{T}, a \in \mathcal{A}.$$

In order to estimate the cost of reading, writing and transferring data, introduce the following weights:

- w_a denotes the average width of attribute a
- f_q denotes the frequency of query q
- $n_{a,q}$ denotes for query q the average number of rows retrieved from or written to the table holding attribute a

Then the cost of reading or writing a in query q is estimated to $W_{a,q} = w_a \cdot f_q \cdot n_{a,q}$ and the cost of transferring attribute a over the network is estimated to $pW_{a,q}$. Notice, that $W_{a,q}$ is only an estimate due to f_q and $n_{a,q}$.

Consider the amount of local data access, A, and let $A = A_{\mathrm{R}} + A_W$ where A_{R} and A_{W} is the amount of read and write access, respectively. For a given site r and query q, A_{R} is the sum of all attribute weights $W_{a,q}$ for which 1) q is a read query, 2) attribute a is stored on r, 3) the transaction that executes query q is executed on r and 4) q accesses any attribute in the table fraction that holds a. As we maintain single-sitedness for reads, $\beta_{a,q}$ can be used to handle 4), resulting in

$$A_{\mathrm{R}} = \sum_{a,t,s,q} W_{a,q}\beta_{a,q}\gamma_{q,t}(1 - \delta_q)x_{t,s}y_{a,s}.$$

Accounting for local access of write queries, A_{W}, is less trivial. Consider the following three approaches:

Access relevant attributes An attribute a at site s should be accounted for if and only if there exists an attribute a' on s that q updates so that a and a' are attributes of the same table. While this accounting is the

most accurate of the three it is also the most expensive as it implies an element of the form $y_{a,s}y_{a',s}$ in the objective function which adds an undesirable amount of $|\mathcal{A}|^2|\mathcal{S}|$ variables and $3|\mathcal{A}|^2|\mathcal{S}|$ constraints to the problem when linearized (see Section 7.2.3).

Access all attributes We can get around the increased complexity by assuming that write queries q always writes to all sites containing table fractions of tables accessed by q, regardless of whether q actually accesses any of the attributes of the fractions. While this is correct for insert statements (assuming that inserts always write complete rows) it is likely an overestimation for updates: imagine a lot of single-attribute updates on a wide table where the above method would have split the attribute in question to a separate partition. This overestimation will imply that the model will partition tables that are updated often or replicate attributes less often than the accounting model described above.

Access no attributes Another approach to simplify the cost function is to completely avoid accounting for local access for writes and solely let the network transfer define the write costs. With this underestimation of write costs, attributes will then tend to be replicated more often than in the first accounting model.

In this chapter we choose the second approach, which gives a conservative overestimate of the write costs as we then obtain more accurate costs for inserts and avoid extending the model with undesirably many variables and constraints. Intuitively speaking, this choice implies that read queries will tend to partition the tables for best possible read-performance, and the write queries will tend to minimize the amount of attribute replication. We now have

$$A_{\mathrm{W}} = \sum_{q,a,s} W_{a,q}\beta_{a,q}\delta_q y_{a,s}$$

and thus

$$A = \sum_{a,t,s,q} W_{a,q}\beta_{a,q}\gamma_{q,t}(1-\delta_q)x_{t,s}y_{a,s} + \sum_{q,a,s} W_{a,q}\beta_{a,q}\delta_q y_{a,s}. \tag{7.3}$$

B accounts for the amount of network transfer and since we enforce single-sitedness for all reads B is solely the sum of transfer costs for write queries. We assume that write queries only transfer the attributes they update and does not transfer to the site that holds their own transaction:

$$B = \sum_{a,t,s,q} W_{a,q}\alpha_{a,q}\gamma_{q,t}\delta_q(1-x_{t,s})y_{a,s}.$$

By noticing that $\sum_{a,t,s,q} \alpha_{a,q}\gamma_{q,t}y_{a,s} = \sum_{a,s,q} \alpha_{a,q}y_{a,s}$ we can construct the minimization problem as

$$\begin{array}{lrll} \min & \sum_{t,a,s} c_1(a,t)x_{t,s}y_{a,s} + \sum_{a,s} c_2(a)y_{a,s} & & \\ \text{s.t.} & \sum_s x_{t,s} & = 1 & \forall t \\ & \sum_s y_{a,s} & \geq 1 & \forall a \\ & y_{a,s} - x_{t,s}\varphi_{a,t} & \geq 0 & \forall a,t \\ & x_{t,s}, y_{a,s} & \in \{0,1\} & \forall t,a,s \end{array} \tag{7.4}$$

where

$$c_1(a,t) = \sum_q W_{q,a}\gamma_{q,t}(\beta_{a,q}(1-\delta_q) - p\alpha_{a,q}\delta_q)$$

and

$$c_2(a) = \sum_q W_{a,q}\delta_q(\beta_{a,q} + p\alpha_{a,q}).$$

Both c_1 and c_2 are completely induced by the schema, query workload and statistics and can therefore be considered static when the partitioning process starts.

7.2.2 Adding load balancing

We are interested in extending the model in (7.4) to also handle load balancing of the sites instead of just minimizing the sum of all data access/transfer. From equation (7.3) define the work of a single site $s \in \mathcal{S}$ as

$$\sum_{a,t} c_3(a,t)x_{t,s}y_{a,s} + \sum_a c_4(a)y_{a,s} \tag{7.5}$$

where $c_3(a,t) = \sum_q W_{a,q}\gamma_{q,t}\beta_{a,q}(1-\delta_q)$ and $c_4(a) = \sum_q W_{a,q}\beta_{a,q}\delta_q$. Introduce the variable m and for each site s let the value of (7.5) be a lower bound for m. Adding m to the objective function is then equivalent to also minimizing the work of the maximally loaded site.

In order to decide how to prioritize cost minimization versus load balancing in the model, introduce a scalar $0 \leq \lambda \leq 1$ and weight the original cost from (7.4) and m by λ and $(1-\lambda)$, respectively. The new objective is then

$$\lambda \sum_{a,t,s} c_1(a,t)x_{t,s}y_{a,s} + \lambda \sum_{a,s} c_2(a)y_{a,s} + (1-\lambda)m \tag{7.6}$$

where m is constrained as follows:

$$\sum_{a,t} c_3(a,t)x_{t,s}y_{a,s} + \sum_{a,q} c_4(a)y_{a,s} \leq m \quad ,\forall s \in \mathcal{S}.$$

Notice that while we are now minimizing (7.6), the objective of (7.4) should still be considered as the actual cost of a solution.

7.2.3 Linerarization

We use the technique discussed in [44, Chapter IV, Theorem 4] to linearize the model. This is done by replacing the quadratic terms in the model with a variable $u_{t,a,s}$ and adding the following new constraints:

$$\begin{array}{ll} u_{t,a,s} \leq x_{t,s} & \forall t, a, s \\ u_{t,a,s} \leq y_{a,s} & \forall t, a, s \\ u_{t,a,s} \geq x_{t,s} + y_{a,s} - 1 & \forall t, a, s \end{array}$$

For $u_{t,a,s} \geq 0$, notice that $u_{t,a,s} = 1$ if and only if $x_{t,s} = y_{a,s} = 1$ and that $u_{t,a,s}$ is guaranteed to be binary if both $x_{t,s}$ and $y_{a,s}$ are binary (thus, there is no need for requiring it explicitly in the model).

Now, the model in (7.4) extended with load balancing looks as follows when linearized:

$$\begin{array}{lrll} \min & \lambda \sum_{t,a,s} c_1(a,t) u_{t,a,s} + \lambda \sum_{a,s} c_2(a) y_{a,s} + (1-\lambda) m & & \\ \text{s.t.} & \sum_s x_{t,s} & = 1 & \forall t \\ & \sum_s y_{a,s} & \geq 1 & \forall a \\ & y_{a,s} - x_{t,s} \varphi_{a,t} & \geq 0 & \forall a, t \\ & \sum_{a,t} c_3(a,t) u_{a,t,s} + \sum_{a,q} c_4(a) y_{a,s} & \leq m & \forall s \\ & u_{t,a,s} - x_{t,s} & \leq 0 & \forall t, a, s \\ & u_{t,a,s} - y_{a,s} & \leq 0 & \forall t, a, s \\ & u_{t,a,s} - x_{t,s} - y_{a,s} + 1 & \geq 0 & \forall t, a, s \\ & x_{t,s}, y_{a,s} & \in \{0,1\} & \forall t, a, s \\ & u_{t,a,s} & \geq 0 & \forall t, a, s \end{array} \tag{7.7}$$

7.2.4 Complexity

The objective function in quadratic programs can be written on the form

$$\tfrac{1}{2} z^{\mathrm{T}} Q z + cz + d$$

where in our case $z = (x_{1,1}, \ldots, x_{|\mathcal{T}|,|\mathcal{S}|}, y_{1,1}, \ldots, x_{|\mathcal{A}|,|\mathcal{S}|})$ is a vector containing the decision variables, Q is a cost matrix, c is a cost vector and d a constant. Q can be easily defined from (7.6) by dividing Q into four quadrants, letting the sub-matrices in the upper-left and lower-right quadrant equal zero and letting the upper-right and lower-left submatrices be defined by $c_1(a,t)$. Q is indefinite and the cost function (7.6) therefore not convex. As shown by Marty and Judice [62] finding optimum when Q is indefinite is NP-hard.

7.3 The SA solver – a heuristic approach

We develop a heuristic based on simulated annealing (see [100]) and will refer to it as the *SA*-solver from now on. The base idea is to alternately fix x and y

and only optimize the not-fixed vector, thereby simplifying the problem. In each iteration we search in the neighborhood of the found solution and accept a worse solution as base for a further search with decreasing probability.

Let $x_{t,s}$ hold an assignment of transactions to sites and define the neighborhood x' of x as a change of location for a subset of the transactions so that for each $t \in \mathcal{T}$ we still have $\sum_s x'_{t,s} = 1$. Similarly, let $y_{a,s}$ hold an assignment of attributes to sites but define the neighborhood y' of y as an extended replication of a subset of the attributes. That is, for each $a \in \mathcal{A}$ in that subset we have $y_{a,s} = 1 \Rightarrow y'_{a,s} = 1$ and $\sum_s y'_{a,s} > \sum_s y_{a,s}$. We found that altering the location for a constant number of 10% of both transactions/attributes yielded the best results. The heuristic now looks as pictured in Algorithm 4. Notice, that the linearization constraints is not

Algorithm 4 The heuristic based on simulated annealing (SA). It iteratively fixes x and y and accepts a worse solution from the neighborhood with decreasing probability.

```
1: Initialize temperature τ > 0 and reduction factor ρ ∈ ]0; 1[
2: Set the number L of inner loops
3: Initialize x randomly so that (7.2) is satisfied
4: fix ← "x"
5: S ← findSolution(fix)
6: while not frozen do
7:     for i ∈ {1, ..., L} do
8:         x ← neighborhood of x
9:         y ← neighborhood of y
10:        S' ← findSolution(fix)
11:        Δ ← cost(S') − cost(S)
12:        p ← a randomly chosen number in [0; 1]
13:        if Δ ≤ 0 or p < e^(−Δ/τ) then
14:            S ← S'
15:        end if
16:        fix ← the element in {"x","y"} \ {fix}
17:    end for
18:    τ ← ρ · τ
19: end while
```

needed since either x or y will be constant in each iteration. This reduces the size of the problem considerably.

7.4 Further improvements

Resonable cuts Consider a table with n attributes together with two queries: one accessing attribute 1 through k and one accessing attribute k

through n. Then it is sufficient to find an optimal distribution for the three attribute groupings $\{1, \ldots, k-1\}$, $\{k\}$ and $\{k+1, \ldots, n\}$, considering each group as an atomic unit and thereby reducing the problem size. In general, it is only necessary to distribute groups of attributes induced by query access overlaps. Chu and Ieong [21] refer to these attribute overlaps as *reasonable cuts*. Even though this will not improve the worst-case complexity, this reduction may still have a large performance impact on some instances.

Bottleneck transactions Also, assuming that transactions follow the 20/80 rule (20% of the transactions generate 80% of the load), the problem can be solved iteratively over $\mathcal{T}$ starting with a small set of the most heavy transactions.

Network latency We can extend the algorithms to also estimate costs of network latency for queries accessing attributes on remote sites. We assume, that all remote access (if any) for queries are done in parallel and with a constant number of requests per query per remote site. Let p_l denote a latency penalty factor and introduce a new binary variable ψ_q for each query q indicating with $\psi_q = 1$ if q accesses any remotely placed attributes. Letting n denote the number of remotely accessed attributes by q we have $n > 0 \Rightarrow \psi_q = 1$ and $n = 0 \Rightarrow \psi_q = 0$, or equivalently $(\psi_q - 1)n = 0$ and $\psi_q - n \leq 0$. This results in the following two classes of new constraints:

$$(\psi_q - 1) \sum_{a,s} \delta_q \alpha_{a,q} \gamma_{q,t} (1 - x_{t,s}) y_{a,s} = 0 \quad , \forall q, t$$

and

$$\psi_q - \sum_{a,s} \delta_q \alpha_{a,q} \gamma_{q,t} (1 - x_{t,s}) y_{a,s} \leq 0 \quad , \forall q, t$$

The total latency in a given partitioning can now be estimated by the sum $p_l \sum_q f_q \psi_q$ which can be added to the cost objective function (7.4).

7.5 Computational results

We assume that the context is a database with a very high transaction count like the memory-only database H-store [91] (now VoltDB[1]) and thus need to compare RAM access versus network transfer time when deciding an appropriate network penalty factor p. A PCI Express 2.0 bus transfers between 32 Gbit/s and 128 Gbit/s while the bandwidth of PC3 DDR3-SDRAM is at least 136 Gbit/s so the bus is the bottleneck in RAM accesses. We assume that the network is well configured and latency is minimal. Therefore the

[1] http://voltdb.com

network penalty factor could be estimated to $p \in [3; 128]$ if either a gigabit or 10-gigabit network is used to connect the physical sites. We assume the use of a 10-gigabit network and therefore set $p = 8$ in our tests unless otherwise stated.

We furthermore mainly focus on minimizing the total costs of execution and therefore set λ low. If λ is kept positive the model will, however, choose the more load balanced layout if there is a cost draw between multiple layouts. We set $\lambda = 0.1$ in our tests unless otherwise stated.

All tests were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo and 4GB 1067 MhZ DDR3 RAM, running Mac OS X 10.5. The GNU Linear Programming Kit[2] (GLPK) 4.39 was used as MIP solver, using only a single thread.

The test implementation is available upon request.

7.5.1 Initial temperature

The temperature τ used in the heuristic described in Section 7.3 determines how willing the algorithm is to accept a worse solution than the currently best found. Let C^* and C denote the objective for the best solution so far and the currently generated solution, respectively. In the computational results provided here we accept a worse solution with 50% probability in the first set of iterations if $\frac{C-C^*}{C} < 5\%$. Referring to the notation used in Algorithm 4, we have $50\% = e^{0.05C^*/\tau}$ and thus an initial temperature of $\tau = -0.05C^*/\ln 0.5$.

7.5.2 The TPC-C v5 instance

We perform tests on the TPC-C version 5.10.1 benchmark[3]. The TPC-C specification describes transactions, queries and database schema but does not provide the statistics needed to create a problem instance. We therefore made some simplified assumptions: all queries are assumed to run with equal frequency and all queries (not transactions) are assumed to access a single row except in the obvious cases where aggregates are used or there are being iterated over the result. In these cases we assume that the query accesses 10 rows. Thereby, the New-Order transaction for example, are assumed to access 11 rows in average.

We model UPDATE queries as two sub-queries: A read-query accessing all the attributes used in the original query and a write-query only accessing the attributes actually being written (and thus whose update needs to be distributed to all replicas).

[2] http://gnu.org/software/glpk
[3] http://www.tpc.org/tpcc

7.5.3 Random instances

To the best of our knowledge there is no standard library of typical OLTP instances with schemas, workloads and statistics so in order to explore the characteristics of the algorithms we perform some experiments on a set of randomly generated instances instead as it showed up to be a considerable administrative and bureaucratic challenge (if possible at all) to collect appropriate instances from "real life" databases. The randomly generated instances vary in several parameters in order to clarify which characteristics that influence the potential cost reduction by applying our vertical partitioning algorithms. The parameters include: number of transactions in workload, number of tables in schema, maximum number of attributes per table, maximum number of queries per transaction, percentage of queries being updates, maximum number of different tables being referred to from a single query, maximum number of individual attributes being referred to by a single query, the set of allowed attribute widths. We define classes of problem instances by upper bounds on all parameters. Individual instances are then generated by choosing the value of each parameter evenly distributed between 1 and its upper bound. That is, if e.g. the maximum allowed number of attributes in tables is k, the number of table attributes for each table in the generated instance will be evenly distributed between 1 and k with a mean of $k/2$.

7.5.4 Results

In the following we perform a series of tests and display the results in tables where each entry holds the found objective of (7.4) for the given instance.

Table 7.1 explores the influence of a set of parameters in the randomly generated instances by varying one parameter at a time while fixing the rest. We test two classes of instances using the SA solver: a smaller with #tables $= |\mathcal{T}| = 20$ and a larger with #tables $= |\mathcal{T}| = 100$. The results suggest that the largest workload reduction is obtained for instances having relatively few queries per transaction, few updates, many attributes per table and/or a moderate number of attribute references per query. The number of table references per query and the allowed attribute widths, however, only seem to have moderate influence on the result.

Table 7.3 compares the QP and SA solvers on the TPC-C benchmark and a set of randomly generated larger instances, divided into two classes with either large or low potential for cost reduction. The random instances are described in Table 7.2 where the columns here refer to the single-letter labels for the parameters shown in Table 7.1. As seen in Table 7.3 the SA solver is generally faster than the QP solver but the QP solver obtains lower costs when the instances are small. Expectedly, the instances in class "rndB..." with many attribute references per query but few queries per

		#tables = $\|\mathcal{T}\| = 20$			#tables = $\|\mathcal{T}\| = 100$		
		$\|\mathcal{S}\| = 1$	$\|\mathcal{S}\| = 2$	$\|\mathcal{S}\| = 3$	$\|\mathcal{S}\| = 1$	$\|\mathcal{S}\| = 2$	$\|\mathcal{S}\| = 3$
A Max queries per transaction	1	0.585	0.309	0.278	3.194	1.784	1.471
	3	1.567	1.478	1.386	5.743	4.550	4.189
	5	1.305	1.054	0.972	8.840	7.569	6.983
B Percent updates queries	0	1.747	1.369	1.110	5.959	4.235	3.510
	10	1.567	1.478	1.386	5.743	4.550	4.189
	30	1.349	1.244	1.263*	5.106	4.555	4.462
C Max attributes per table	5	0.520	0.520*	0.520*	2.583	2.772*	2.712*
	15	1.567	1.478	1.386	5.743	4.550	4.189
	35	1.643	0.968	0.850	14.970	7.341	5.355
D Max table references per query	2	0.602	0.430	0.356	3.447	3.022	2.865
	5	1.567	1.478	1.386	5.743	4.550	4.189
	10	2.246	1.607	1.516	8.147	6.063	5.623
E Max attribute references per query	5	0.678	0.288	0.199	5.176	2.526	1.969
	15	1.567	1.478	1.386	5.743	4.550	4.189
	25	1.115	0.988	1.008*	5.641	5.909*	5.684*
F Allowed attribute widths	$\{2, 4, 8\}$	1.194	1.080	1.030	4.456	3.488	3.500*
	$\{\mathbf{4}, \mathbf{8}\}$	1.567	1.478	1.386	5.743	4.550	4.189
	$\{4, 8, 16\}$	2.387	2.160	2.060	8.912	6.977	7.000

Table 7.1: Comparing the effect of parameter changes. Results were found using the SA solver. We test three possible values for each parameter, varying one parameter at the time and fixing all other parameters at their default value (marked with bold). The costs are shown in units of 10^6. Tests are divided into two classes having both the number of transactions and schema tables equal to 20 (left) and 100 (right), respectively. The results suggest that the largest workload reduction, unsurprisingly, is obtained for instances having relatively few queries per transaction, few updates, many attributes per table and/or a moderate number of attribute references per query. The number of table references per query and the allowed attribute widths, however, only seem to have moderate influence on the result.

Name	A	B	C	D	E	F	$\lvert\mathcal{T}\rvert$	#tables
rndAt4x15	3	10	30	3	8	$\{2, 4, 8, 16\}$	15	4
rndAt8x15	3	10	30	3	8	$\{2, 4, 8, 16\}$	15	8
rndAt8x15u50	3	50	30	3	8	$\{2, 4, 8, 16\}$	15	8
rndAt16x15	3	10	30	3	8	$\{2, 4, 8, 16\}$	15	16
rndAt32x15	3	10	30	3	8	$\{2, 4, 8, 16\}$	15	32
rndAt4x100	3	10	30	3	8	$\{2, 4, 8, 16\}$	100	4
rndAt8x100	3	10	30	3	8	$\{2, 4, 8, 16\}$	100	8
rndAt16x100	3	10	30	3	8	$\{2, 4, 8, 16\}$	100	16
rndAt32x100	3	10	30	3	8	$\{2, 4, 8, 16\}$	100	32
rndBt4x15	3	10	5	6	28	$\{2, 4, 8, 16\}$	15	4
rndBt8x15	3	10	5	6	28	$\{2, 4, 8, 16\}$	15	8
rndBt16x15	3	10	5	6	28	$\{2, 4, 8, 16\}$	15	16
rndBt16x15u50	3	50	5	6	28	$\{2, 4, 8, 16\}$	15	16
rndBt32x15	3	10	5	6	28	$\{2, 4, 8, 16\}$	15	32
rndBt4x100	3	10	5	6	28	$\{2, 4, 8, 16\}$	100	4
rndBt8x100	3	10	5	6	28	$\{2, 4, 8, 16\}$	100	8
rndBt16x100	3	10	5	6	28	$\{2, 4, 8, 16\}$	100	16
rndBt32x100	3	10	5	6	28	$\{2, 4, 8, 16\}$	100	32

Table 7.2: Random instances used when comparing the QP and SA solvers in Table 7.3. The instances in the upper part (rndA...) are expected to get a large cost reduction while instances in the lower part (rndB...) are expected to get a small cost reduction. The columns refer to the single-letter labels for the parameters shown in Table 7.1.

table gains little or no cost reduction by applying the algorithms. TPC-C, on the other hand, gets a cost reduction of 37% and the random instances in class "rndA...", with many attributes per table and relatively few attribute references per query, get a cost reduction between 25% and 85%. None of the algorithms found a cost reduction for the instances rndAt4x100 and rndAt8x100 because of the "overweight" of transactions compared to the number of attributes in the schemas.

Table 7.4 depicts an actual partitioning of TPC-C constructed by the QP solver for three sites.

Table 7.5 illustrates the effect of disjoint versus nondisjoint partitioning, that is, partitioning without and with attribute replication. As seen, greater cost reduction can be obtained when allowing replication but in exchange to increased computation time.

Table 7.6 compares two different kinds of partition placements: 1) all partitions being located at one single site (thereby avoiding inter-site transfers) and 2) partitions being located at remote sites. These two situations can be simulated by setting $p = 0$ and $p > 0$, respectively. The benefits of local placements are given by the amount of updates in the workload as only updates cause inter-site transfers. More updates implies larger costs for remote placements. For a somewhat extreme case, instance "rndAt8x15u50", with 50% of the queries being updates, the costs are about 33% lower when

				QP		SA		
Instance	$\|\mathcal{A}\|$	$\|\mathcal{T}\|$	$\|\mathcal{S}\|$	Cost	Time (s)	Cost	Time (s)	$\|\mathcal{S}\| = 1$
TPC-C v5	92	5	2	**0.133**	1	0.138	5	0.208
TPC-C v5	92	5	3	**0.132**	6	**0.132**	5	0.208
TPC-C v5	92	5	4	**0.132**	33	**0.132**	5	0.208
rndAt4x15	54	15	4	**(0.332)**	1800	0.396	10	0.933
rndAt8x15	105	15	4	**(0.324)**	1800	0.327	18	0.808
rndAt16x15	225	15	4	**(0.267)**	1800	0.309	41	1.180
rndAt32x15	492	15	4	(0.315)	1800	**0.217**	89	1.491
rndAt64x15	1023	15	4	**(0.269)**	1800	0.268	190	1.452
rndAt4x100	54	100	4	(8.001)	1800	8.246	79	**7.946**
rndAt8x100	105	100	4	(7.681)	1800	8.018	150	**7.454**
rndAt16x100	225	100	4	-	t/o	**6.525**	321	8.741
rndAt32x100	492	100	4	-	t/o	**4.501**	728	8.916
rndAt64x100	1023	100	4	-	t/o	**4.119**	1531	9.591
rndBt4x15	12	15	4	0.303	65	0.303	3	**0.303**
rndBt8x15	27	15	4	(0.448)	1800	**0.424**	6	0.440
rndBt16x15	49	15	4	**(0.333)**	1800	0.334	9	0.385
rndBt32x15	98	15	4	**(0.319)**	1800	**0.319**	16	0.361
rndBt64x15	210	15	4	**(0.221)**	1800	**0.221**	31	0.229
rndBt4x100	54	100	4	(4.484)	1800	**2.251**	18	**2.251**
rndBt8x100	105	100	4	(4.323)	1800	**2.419**	37	**2.419**
rndBt16x100	225	100	4	(2.001)	1800	**1.774**	62	**1.774**
rndBt32x100	492	100	4	(2.419)	1800	**1.999**	124	**1.999**
rndBt64x100	1023	100	4	-	1800	**2.473**	270	**2.473**

Table 7.3: Comparing the QP algorithm with the simulated annealing based heuristic (SA), allowing attribute replication and with remote partition placement. Costs are shown in units of 10^6. The SA algorithm had a 30 second time limit for each iteration and if the limit was reached it proceeded with another neighborhood. The QP algorithm had a time bound of 30 minutes and an MIP tolerance gap of 0.1%. Where the time limit was reached, the best found cost (if any) is written in parentheses. "t/o" indicates that no integer solution was found within the time limit.

Site 1
Transaction *Payment*
Customer.C_BALANCE
Customer.C_CITY
Customer.C_CREDIT
Customer.C_CREDIT_LIM
Customer.C_DATA
Customer.C_DISCOUNT
Customer.C_D_ID
Customer.C_FIRST
Customer.C_ID
Customer.C_LAST
Customer.C_MIDDLE
Customer.C_PHONE
Customer.C_SINCE
Customer.C_STATE
Customer.C_STREET_1
Customer.C_STREET_2
Customer.C_W_ID
Customer.C_ZIP
District.D_CITY
District.D_ID
District.D_NAME
District.D_STATE
District.D_STREET_1
District.D_STREET_2
District.D_W_ID
District.D_YTD
District.D_ZIP
History.H_AMOUNT
History.H_C_D_ID
History.H_C_ID
History.H_C_W_ID
History.H_DATA
History.H_DATE
History.H_D_ID
History.H_W_ID
OrderLine.OL_DIST_INFO
OrderLine.OL_NUMBER
Stock.S_ORDER_CNT
Stock.S_REMOTE_CNT
Stock.S_YTD
Warehouse.W_CITY
Warehouse.W_ID
Warehouse.W_NAME
Warehouse.W_STREET_1
Warehouse.W_STREET_2
Warehouse.W_YTD
Warehouse.W_ZIP

Site 2
Transaction *StockLevel*
Customer.C_CITY
Customer.C_DELIVERY_CNT
Customer.C_PAYMENT_CNT
Customer.C_SINCE
Customer.C_YTD_PAYMENT
District.D_ID
District.D_NEXT_O_ID
District.D_W_ID
Item.I_IM_ID
OrderLine.OL_D_ID
OrderLine.OL_I_ID
OrderLine.OL_O_ID
OrderLine.OL_W_ID
Stock.S_I_ID
Stock.S_QUANTITY
Stock.S_W_ID

Site 3
Transaction *Delivery*
Transaction *NewOrder*
Transaction *OrderStatus*
Customer.C_BALANCE
Customer.C_CREDIT
Customer.C_DISCOUNT
Customer.C_D_ID
Customer.C_FIRST
Customer.C_ID
Customer.C_LAST
Customer.C_MIDDLE
Customer.C_W_ID
District.D_ID
District.D_NEXT_O_ID
District.D_TAX
District.D_W_ID
Item.I_DATA
Item.I_ID
Item.I_NAME
Item.I_PRICE
NewOrder.NO_D_ID
NewOrder.NO_O_ID
NewOrder.NO_W_ID
Order.O_ALL_LOCAL
Order.O_CARRIER_ID
Order.O_C_ID
Order.O_D_ID
Order.O_ENTRY_D
Order.O_ID
Order.O_OL_CNT
Order.O_W_ID
OrderLine.OL_AMOUNT
OrderLine.OL_DELIVERY_D
OrderLine.OL_D_ID
OrderLine.OL_I_ID
OrderLine.OL_O_ID
OrderLine.OL_QUANTITY
OrderLine.OL_SUPPLY_W_ID
OrderLine.OL_W_ID
Stock.S_DATA
Stock.S_DIST_01
Stock.S_DIST_02
Stock.S_DIST_03
Stock.S_DIST_04
Stock.S_DIST_05
Stock.S_DIST_06
Stock.S_DIST_07
Stock.S_DIST_08
Stock.S_DIST_09
Stock.S_DIST_10
Stock.S_I_ID
Stock.S_QUANTITY
Stock.S_W_ID
Warehouse.W_ID
Warehouse.W_TAX

Table 7.4: The result of a vertical partitioning of the TPC-C benchmark using the QP solver for three sites. Each column represents the contents of a site and is divided into three sub-sections: a header, a section holding the transaction names and a longer section holding the attributes assigned to the respective site.

placing the partitions locally.

7.6 Conclusion

We have constructed a cost model for vertical partitioning of relational OLTP databases together with a quadratic integer program that distributes both attributes and transactions to a set of sites while allowing attribute replication, preserving single-sitedness for read queries and in which load balancing vs. total cost minimization can be prioritized arbitrarily.

We also presented a more scalable heuristic which seems to deliver good results. For both algorithms we obtained a cost reduction of 37% in our model of TPC-C and promising results for the random instances. Even though the latter theoretically can be constructed with arbitrary high/low benefits from vertical partitioning, the test runs on our selected subset of random instances seem to indicate that 1) our heuristic scales far better than the QP-solver, and 2) it can obtain valuable cost reductions on many real-world OLTP databases, as we tried to select the parameters realistically.

One thing we miss, however, is an official OLTP testbed – a library containing realistic OLTP workloads, schemas and statistics. Such a collection of realistic instances could serve as base for several insteresting and important studies for understanding the nature and characteristics of OLTP databases.

Acknowledgements. The author would like to acknowledge Daniel Abadi for competent and valuable discussions and feedback. Also, Rasmus Pagh, Philippe Bonnet and Laurent Flindt Muller have been very helpful with insightful comments on preliminary versions of the text.

Instance	$\|\mathcal{A}\|$	$\|\mathcal{T}\|$	$\|\mathcal{S}\|$	w. replication Cost	w. replication Time (s)	w/o replication Cost	w/o replication Time (s)	Ratio
TPC-C v5	92	5	1	0.208	0	0.208	0	-
TPC-C v5	92	5	2	0.133	1	0.207	1	64%
TPC-C v5	92	5	3	0.132	6	0.207	2	64%
TPC-C v5	92	5	4	0.132	33	0.207	3	64%
rndAt4x15	54	15	2	4.855	28	6.799	1	71%
rndAt8x15	105	15	2	4.710	517	5.809	6	81%
rndAt8x15	27	15	2	4.244	4	4.402	0	96%
rndAt16x15	49	15	2	3.410	34	3.852	0	89%

Table 7.5: Computational results from solving the TPC-C benchmark and a few random instances with the QP solver. Costs are shown in units of 10^5. The table shows that costs can be reduced by allowing attribute replication and that TPC-C does not benefit noticeably from being partitioned and distributed to more than two sites. The *Ratio* column displays the ratio between the replicated and non-replicated cost.

Instance	$\|\mathcal{A}\|$	$\|\mathcal{T}\|$	$\|\mathcal{S}\|$	Local Cost (QP)	Local Cost (SA)	Remote Cost (QP)	Remote Cost (SA)
TPC-C v5	92	5	1	1.916	1.916	1.916	1.916
TPC-C v5	92	5	2	1.210	1.208	1.221	1.273
TPC-C v5	92	5	3	1.208	1.208	1.220	1.220
rndAt4x15	54	15	2	4.709	4.742	4.855	4.888
rndAt8x15	105	15	2	4.424	4.808	4.710	5.187
rndAt8x15u50	105	20	2	3.189	3.313	4.778	4.873
rndBt8x15	27	15	2	4.365	4.332	4.244	4.730
rndBt16x15	49	15	2	3.335	3.387	3.410	3.404
rndBt16x15u50	49	20	2	5.066	5.220	5.438	5.438

Table 7.6: Comparing the costs of local ($p = 0$) versus remote ($p > 0$) location of partitions and with attribute replication allowed. Costs are in units of 10^5. Write-rarely instances or instances in class "rndB..." do not benefit noticeably by placing all partitions locally, even the instances with 50% update queries, however instances in class "rndA..." with a large update ratio do. The reason is that only updates cause inter-site transfer. That the costs of the local placement for rndBt8x15 is *larger* than when placed remotely is since $\lambda > 0$.

Bibliography

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, volume 28(2) of *SIGMOD Record*, pages 275–286. ACM, 1999.

[2] R. Agarwal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, Jan 2001.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988. ISSN 0001-0782.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB '94)*, pages 487–499. Morgan Kaufmann Publishers, 1994.

[5] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *Proceedings of the 2004 ACM SIGMOD international ...*, Jan 2004.

[6] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[7] R. R. Amossen. Vertical partitioning of relational oltp databases using integer programming. In *Proceedings of the ICDE Workshops 2010*, pages 93–98, mar. 2010.

[8] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory (ICDT '09)*, pages 121–126. ACM, 2009. ISBN 978-1-60558-423-2.

[9] R. R. Amossen, A. Campagna, and R. Pagh. Better size estimation for sparse matrix products. In M. Serna, R. Shaltiel,

K. Jansen, and J. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 6302 of *Lecture Notes in Computer Science*, pages 406–419. Springer Berlin / Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-15369-3_31.

[10] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM '02)*, pages 1–10. Springer-Verlag, 2002. ISBN 3-540-44147-6.

[11] Benchmarks. Computer language benchmark game. http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=all.

[12] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In T. Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer, 2007. ISBN 978-3-540-77118-0.

[13] D. Bini, M. Capovani, F. Romani, and G. Lotti. $\mathcal{O}(n^{2.77})$ complexity for $n \times n$ approximate matrix multiplication. *Inform. Process. Lett. 8 no. 5*, pages 234–235, 1979.

[14] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005. URL http://www.cidrdb.org/cidr2005/papers/P19.pdf.

[15] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI '03, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003. URL http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-90/borgelt.pdf.

[16] C. Borgelt. Recursion pruning for the apriori algorithm. In *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. URL http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-126/borgelt.pdf.

[17] C. Borgelt. An implementation of the fp-growth algorithm. In *OSDM '05: Proceedings of the 1st international workshop on open source data*

mining, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-210-0. doi: http://doi.acm.org/10.1145/1133905.1133907.

[18] S. Chakravarthy, J. Muthuraj, and R. Varadarajan. An objective function for vertically partitioning relations in distributed databases and its .... *Distributed and parallel databases*, Jan 1994.

[19] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS '00)*, pages 268–279. ACM, 2000.

[20] F. Chin. An o (n) algorithm for determining a near-optimal computation order of matrix chain products. *Communications of the ACM*, Jan 1978.

[21] W. Chu and I. Ieong. A transaction-based approach to vertical partitioning forrelational database systems. *IEEE Transactions on Software Engineering*, Jan 1993.

[22] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, Dec. 1997.

[23] E. Cohen. Structure prediction and computation of sparse matrix products. *J. Comb. Optim*, 2(4):307–332, 1998.

[24] D. Coppersmith. Rectangular matrix multiplication revisited. *J. Complex.*, 13(1):42–49, 1997. ISSN 0885-064X. doi: http://dx.doi.org/10.1006/jcom.1997.0438.

[25] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 82–90, 1981.

[26] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7.

[27] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 82–90, 2008.

[28] D. W. Cornell and P. S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Trans. Softw. Eng.*, 16(2):248–258, 1990. ISSN 0098-5589.

[29] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *soda00*, pages 743–752, 2000. URL `http://doi.acm.org/10.1145/338219.338634`.

[30] M. Dietzfelbinger and F. M. auf der Heide. Simple, efficient shared memory simulations. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 110–119, Velen, Germany, June 30–July 2, 1993. SIGACT and SIGARCH. Extended abstract.

[31] D. Dor and U. Zwick. Selecting the median. In *Proceedings of the 6th annual ACM-SIAM Symposium on Discrete algorithms (SODA '95)*, pages 28–37. SIAM, 1995. ISBN 0-89871-349-8.

[32] D. Dubhashi, D. Dubhashi, D. Ranjan, and D. Ranjan. Balls and bins: A study in negative dependence. *Random Structures & Algorithms*, 13:99–124, 1996.

[33] M. Eisner. Mathematical techniques for efficient record segmentation in large shared databases. *Journal of the Assoclauon for Computing Machinery*, Jan 1976.

[34] J. Erickson. Tail bounds, October 2005. URL `http://www.cs.uiuc.edu/class/fa05/cs473g/lectures/10-tailbounds.pdf`. University of Illinois, course CS473G notes, lecture 10.

[35] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3. doi: http://dx.doi.org/10.1109/SC.2004.26.

[36] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo. Frequent itemset mining on graphics processors. In P. A. Boncz and K. A. Ross, editors, *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, pages 34–42. ACM, 2009. ISBN 978-1-60558-701-1. URL `http://doi.acm.org/10.1145/1565694.1565702`.

[37] S. Ganguly and B. Saha. On estimating path aggregates over streaming graphs. In *Proceedings of 17th International Symposium on Algorithms and Computation, (ISAAC '06)*, volume 4288 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2006. ISBN 3-540-49694-7.

[38] S. Ganguly, M. Garofalakis, A. Kumar, and R. Rastogi. Join-distinct aggregate estimation over update streams. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems (PODS '05)*, pages 259–270. ACM, 2005. ISBN 1-59593-062-0.

[39] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. D. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 577–588. ACM, 2005. ISBN 1-59593-154-6; 1-59593-177-5. URL `http://www.vldb2005.org/program/paper/thu/p577-ghoting.pdf`.

[40] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 541–550. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-804-4. URL `http://www.vldb.org/conf/2001/P541.pdf`.

[41] S. S. Godbole. On efficient computation of matrix chain products. *IEEE Trans. Comput.*, 22:864–866, September 1973. ISSN 0018-9340.

[42] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTerasort: high performance graphics co-processor sorting for large database management. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 325–336. ACM, 2006. ISBN 1-59593-256-9. URL `http://doi.acm.org/10.1145/1142473.1142511`.

[43] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/355791.355796.

[44] P. L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer Verlag, 1968. ISBN 0-387-04291-1.

[45] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov*, 8(1):53–87, 2004. URL `http://dx.doi.org/10.1023/B:DAMI.0000005258.31418.83`.

[46] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In

SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 511–524, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: http://doi.acm.org/10.1145/1376616.1376670.

[47] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961. ISSN 0001-0782.

[48] T. C. Hu and M. T. Shin. Computation of matrix chain products. part i. *SIAM J. COMPUT*, Jan 1982.

[49] T. C. Hu and M. T. Shin. Computation of matrix chain products. part ii. *SIAM J. COMPUT.*, Jan 1984.

[50] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *J. Complex.*, 14(2):257–299, 1998. ISSN 0885-064X. doi: http://dx.doi.org/10.1006/jcom.1998.0476.

[51] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall Advanced REference Series, Englewood Cliffs, NJ, 1988.

[52] T. Johnson. Performance measurements of compressed bitmap indices. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 278–289, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7.

[53] W. M. Jr, P. Schweitzer, and T. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, Jan 1972.

[54] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[55] Klyuvev, V.V., and N. I. Kokovkin-Shcherbak. On the minimization of the number of arithmetic operations for the solution of linear algebraic systems of equations. Technical Report CS 24, Computer Science Dept., Stanford University, 1965.

[56] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *arXiv*, cs.DS, Nov 2010.

[57] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *24th International Parallel and Distributed Processing (IPDPS '10)*. IEEE, 2010.

[58] E. Li and L. Liu. Optimization of frequent itemset mining on multiple-core processor. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1275–1285. ACM, 2007. ISBN 978-1-59593-649-3. URL `http://www.vldb.org/conf/2007/papers/industrial/p1275-liu.pdf`.

[59] P. Lilly. From voodoo to geforce: The awsome history of 3d graphics, May 2009. URL `http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective`.

[60] A. Lingas. A fast output-sensitive algorithm for boolean matrix multiplication. In *Proceedings of the 17th European Symposium on Algorithms (ESA '09)*, volume 5757 of *Lecture Notes in Computer Science*, pages 408–419. Springer, 2009. ISBN 978-3-642-04127-3. URL `http://dx.doi.org/10.1007/978-3-642-04128-0`.

[61] W. Liu, Muller-Wittig, and B. Schmidt. Performance predictions for general-purpose computation on gpus. pages 50 –50, sep. 2007. doi: 10.1109/ICPP.2007.67.

[62] K. G. Marty and J. Judice. On the complexity of finding stationary points of nonconvex quadratic programs. *Opsearch*, 33(3):162–166, 1996.

[63] S. Menon. Allocating fragments in distributed databases. *IEEE transactions on parallel and distributed systems*, Jan 2005.

[64] A. R. Meyer. Expectation & variance, May 2006. URL `http://www.cs.princeton.edu/courses/archive/fall06/cos341/handouts/variance-notes.pdf`. Massachusetts Institute of Technology, course notes week 13.

[65] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. ISBN 0-521-47465-5.

[66] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4): 680–710, December 1984. ISSN 0362-5915.

[67] S. B. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *SIGMOD Rec.*, 18(2):440–450, 1989. ISSN 0163-5808.

[68] NVIDIA. Nvidia opencl best practices guide. 2009. URL http://developer.download.nvidia.com/compute/cuda/2_3/opencl/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[69] C. Ong, M. Weldon, D. Cyca, and M. Okoniewski. Acceleration of large-scale fdtd simulations on high performance gpu clusters. pages 1 –4, jun. 2009. doi: 10.1109/APS.2009.5171722.

[70] Oracle. A technical overview of the sun oracle exadata storage server and database machine. White paper, September 2009. URL http://www.oracle.com/us/solutions/datawarehousing/039572.pdf.

[71] Oracle. Oracle exadata database machine overview, 2010. URL http://www.oracle.com/ocom/groups/public/@otn/documents/webcontent/128045.pdf.

[72] Oracle. Exadata price list, September 2010. URL http://www.oracle.com/us/corporate/pricing/exadata-pricelist-070598.pdf.

[73] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. 2007. ISSN 1467-8659. URL http://www.blackwell-synergy.com/doi/abs/10.1111/j.1467-8659.2007.01012.x.

[74] A. Pagh and R. Pagh. Scalable computation of acyclic joins. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 225–232, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-318-2.

[75] R. Pagh. Reduction of triangle reporting to boolean matrix multiplication, 2009. Personal communication.

[76] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[77] V. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. *Foundations of Computer Science, 1978., 19th Annual Symposium on DOI - 10.1109/SFCS.1978.34*, pages 166–176, 1978.

[78] V. Pan. How can we speed up matrix multiplication? *SIAM review*, 26(3):393–415, 1984.

[79] V. Y. Pan. New combinations of methods for the acceleration of matrix multiplications. *Computers & Mathematics with Applications*, 7(1):73 – 125, 1981. ISSN 0898-1221. doi: DOI: 10.1016/0898-1221(81)90009-2.

[80] S. Pramanik and D. Vineyard. Optimizing join queries in distributed databases. *Software Engineering, IEEE Transactions on*, 14(9):1319 –1326, Sept. 1988. ISSN 0098-5589.

[81] B. Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In R. J. B. Jr., B. Goethals, and M. J. Zaki, editors, *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. URL `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-126/racz.pdf`.

[82] B. Rácz, F. Bodon, and L. Schmidt-Thieme. On benchmarking frequent itemset mining algorithms: from measurement to analysis. In *OSDM '05: Proceedings of the 1st international workshop on open source data mining*, pages 36–45, New York, NY, USA, 2005. ACM. ISBN 1-59593-210-0. doi: http://doi.acm.org/10.1145/1133905.1133911.

[83] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)*, Jan 1985.

[84] N. Santhanam. Random variables: variance, March 2009. URL `http://www-ee.eng.hawaii.edu/~prasadsn/11.%20variance.pdf`.

[85] R. Sarathy, B. Shetty, and A. Sen. A constrained nonlinear 0-1 program for data allocation. *European Journal of Operational Research*, 102(3):626–647, November 1997.

[86] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. *Experimental and Efficient Algorithms*, pages 606–609, 2005.

[87] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981. doi: 10.1137/0210032. URL `http://link.aip.org/link/?SMJ/10/434/1`.

[88] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[89] J. Son and M. Kim. An adaptable vertical partitioning method in distributed systems. *The Journal of Systems & Software*, Jan 2004.

[90] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, May 1984.

[91] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, Vienna, Austria, 2007.

[92] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[93] T. Uno, M. Kiyomi, and H. Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In R. J. B. Jr., B. Goethals, and M. J. Zaki, editors, *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. URL `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-126/uno.pdf`.

[94] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, Jan. 1987.

[95] Wikipedia. Graphics processing unit, 2010. URL `http://en.wikipedia.org/wiki/Graphics_processing_unit`.

[96] D. E. Willard. Efficient processing of relational calculus expressions using range query theory. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 164–175, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8.

[97] D. E. Willard. Quasilinear algorithms for processing relational calculus expressions (preliminary report). In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 243–257, New York, NY, USA, 1990. ACM. ISBN 0-89791-352-3.

[98] V. Williams and R. Williams. Triangle detection versus matrix multiplication: A study of truly subcubic reducibility.

[99] S. Winograd. A new algorithm for inner product. *IEEE Trans. Comput.*, 17(7):693–694, 1968. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1968.227420.

[100] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998. ISBN 0-471-28366-5.

[101] K. Wu, E. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices, 2006. URL `http://citeseer.ist.psu.edu/article/wu04efficient.html`.

[102] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[103] Y. Ye and C.-C. Chiang. A parallel apriori algorithm for frequent itemsets mining. In *SERA*, pages 87–94. IEEE Computer Society, 2006. ISBN 0-7695-2656-X. URL http://doi.ieeecomputersociety.org/10.1109/SERA.2006.6.

[104] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005. ISSN 1549-6325. doi: http://doi.acm.org/10.1145/1077464.1077466.

[105] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, 20, 1997.

[106] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, Oct./Dec. 1999. ISSN 1092-3063. URL http://www.math.utah.edu/∼beebe.

[107] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 59, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: http://dx.doi.org/10.1109/ICDE.2006.150.