Work-Efficient Query Evaluation with PRAMs

Jens Keppeler
TU Dortmund University, Germany

Thomas Schwentick
TU Dortmund University, Germany

Christopher Spinrath
TU Dortmund University, Germany

Abstract

The paper studies query evaluation in parallel constant time in the PRAM model. While it is well-known that all relational algebra queries can be evaluated in constant time on an appropriate CRCW-PRAM, this paper is interested in the efficiency of evaluation algorithms, that is, in the number of processors or, asymptotically equivalent, in the work. Naive evaluation in the parallel setting results in huge (polynomial) bounds on the work of such algorithms and in presentations of the result sets that can be extremely scattered in memory. The paper first discusses some obstacles for constant time PRAM query evaluation. It presents algorithms for relational operators that are considerably more efficient than the naive approaches. Further it explores three settings, in which efficient sequential query evaluation algorithms exist: acyclic queries, semi-join algebra queries, and join queries – the latter in the worst-case optimal framework. Under natural assumptions on the representation of the database, the work of the given algorithms matches the best sequential algorithms in the case of semi-join queries, and it comes close in the other two settings. An important tool is the compaction technique from Hagerup (1992).

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Database query processing and optimization (theory)

Keywords and phrases PRAM, query evaluation, work-efficient, parallel, acyclic queries, free-connex queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.16

Related Version The full version of this paper contains some proofs and additional details omitted in this version due to space constraints.
Full Version: `https://arxiv.org/abs/2301.08178` [19]

Acknowledgements We are grateful to Uri Zwick for clarifications regarding results in [13] and to Jonas Schmidt and Jennifer Todtenhoefer for careful proofreading. We thank Martin Dietzfelbinger for helpful discussions. Furthermore, we thank the reviewers of ICDT for many insightful suggestions.

1 Introduction

Parallel query evaluation has been an active research area during the last 10+ years. Parallel evaluation algorithms have been thoroughly investigated, mostly using the Massively Parallel Communication (MPC) model [7]. For surveys we refer to [20, 15].

The MPC model is arguably very well-suited to study parallel query evaluation. However, it is not the only model for parallel query evaluation. Indeed, there is also the Parallel Random Access Machine (PRAM) model, a more "theoretical" model which allows for a more fine-grained analysis of parallel algorithms, particularly in non-distributed settings. It was shown by Immerman [16, 17] that PRAMs with polynomially many processors can evaluate first-order formulas and thus relational algebra queries in time $\mathcal{O}(1)$.

In the study of PRAM algorithms it is considered very important that algorithms perform well compared with sequential algorithms. The overall number of computation steps in a PRAM-computation is called its *work*. An important goal is to design parallel algorithms that are *work-optimal* in the sense that their work asymptotically matches the running time of the best sequential algorithms.

Obviously, for $\mathcal{O}(1)$-time PRAM algorithms the work and the number of processors differ at most by a constant factor. Thus, the result by Immerman shows that relational algebra queries can be evaluated with polynomial work. Surprisingly, to the best of our knowledge, work-efficiency of $\mathcal{O}(1)$-time PRAM algorithms for query evaluation has not been investigated in the literature. This paper is meant to lay some groundwork in this direction.

The proof of the afore-mentioned result that each relational algebra query can be evaluated in constant-time by a PRAM with polynomial work is scattered over several papers. It consists basically of three steps, translating from queries to first-order logic formulas [11], to bounded-depth circuits [6], and then to PRAMs [16]. It was not meant as a "practical translation" of queries and does not yield one. However, it is not hard to see directly that the operators of the relational algebra can, in principle, be evaluated in constant time on a PRAM. It is a bit less obvious, though, how the output of such an operation is represented, and how it can be fed into the next operator.

▶ **Example 1.1.** Let us consider an example to illustrate some issues of constant-time query evaluation on a PRAM. Let q be the following conjunctive query, written in a rule-based fashion, for readability.

$$q(x,y,z) \leftarrow E(x,x_1), E(x_1,x_2), E(y,y_1), E(y_1,y_2), E(z,z_1), E(z_1,z_2), R(x_2,y_2,z_2)$$

A (very) naive evaluation algorithm can assign one processor to each combination of six E-tuples and one R-tuple, resulting in work $\mathcal{O}(|E|^6|R|)$. Since the query is obviously acyclic, it can be evaluated more efficiently in the spirit of Yannakakis' algorithm. For simplicity, we ignore the semi-join step of the Yannakakis algorithm. The join underlying the first two atoms $E(x,x_1), E(x_1,x_2)$ can be computed as a sub-query $q_1(x,x_2) \leftarrow E(x,x_1), E(x_1,x_2)$. This can be evaluated by $|E|^2$ many processors, each getting a pair of tuples $E(a_1,a_2), E(b_1,b_2)$ and producing output tuple (a_1,b_2) in case $a_2 = b_1$. The output can be written into a 2-dimensional table, indexed in both dimensions by the tuples of E. In the next round, the output tuples can be joined with tuples from R to compute $q_2(x,y_2,z_2) \leftarrow E(x,x_1), E(x_1,x_2), R(x_2,y_2,z_2)$. However, since it is not clear in advance, which entries of the two-dimensional table carry q_1-tuples, the required work is about $|E|^2 \cdot |R|$. Output tuples of q_2 can again be written into a 2-dimensional table, this time indexed by one tuple from E (for x) and one tuple from R (for y_2 and z_2). Proceeding in a similar fashion, q can be evaluated with work $\mathcal{O}(|E|^3|R|)$. Other evaluation orders are possible but result in similar work bounds. In terms of the input size of the database, this amounts to $\mathcal{O}(\mathsf{IN}^4)$, whereas Yannakakis' algorithm yields $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})$ in the sequential setting, where IN denotes the number of tuples in the given relations and OUT the size of the query result, respectively.

Let us take a look at the representation of the output of this algorithm. The result tuples reside in a 3-dimensional table, each indexed by a tuple from E. It is thus scattered over a space of size $|E|^3$, no matter the size of the result. Furthermore, the same output tuple might occur several times due to several valuations. To produce a table, in which each output tuples occurs exactly once, a deduplication step is needed, which could yield additional work in the order of $|E|^6$.

The example illustrates two challenges posed by the $\mathcal{O}(1)$-time PRAM setting, which will be discussed in more detail in Section 3.

- It is, in general, not possible to represent the result of a query in a compact form, say, as an array, contiguously filled with result tuples. This obstacle results here in upper bounds in terms of the size of the input database, but not in the size of the query result.
- It might be necessary to deduplicate output (or intermediate) relations, but, unfortunately, this cannot be done by sorting a relation, since sorting is not possible in $\mathcal{O}(1)$-time on a PRAM, either.

We will use compactification techniques for PRAMs from [14] to deal with the first challenge. The second challenge is addressed by a suitable representation of the relations. Besides the setting without any assumptions, we consider the setting, where data items are mapped to an initial segment of the natural numbers by a dictionary, and the setting where the relations are represented by ordered arrays.

We show that, for each $\varepsilon > 0$, there is a Yannakakis-based algorithms for acyclic join queries in the dictionary setting with an upper work bound of $\mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}$. Two other results are work-optimal algorithms for queries of the semijoin algebra and almost worst-case and work-optimal algorithms for join queries.

We emphasize that the stated result does not claim a fixed algorithm that has an upper work bound W such that, for every $\varepsilon > 0$, it holds $W \in \mathcal{O}(\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon}$. It rather means that there is a uniform algorithm that has ε as a parameter and has the stated work bound, for each fixed $\varepsilon > 0$. The linear factor hidden in the $\mathcal{O}$-notation thus depends on ε. This holds analogously for our other upper bounds of this form.

This paper consists roughly of three parts, each of which has some contributions to our knowledge on work-efficient constant-time PRAM query evaluation.

The first part presents some preliminaries in Section 2, discusses the framework including some of our (typical) assumptions and data structures in Section 4, surveys lower bound results that pose challenges for $\mathcal{O}(1)$-time PRAM query evaluation in Section 3, and presents some basic operations that will be used in our query evaluation algorithms in Section 4.

The second part presents algorithms for these basic operations in Section 5 and for relational operators in Section 6.

After this preparation, the third part studies query evaluation in three settings in which (arguably) efficient algorithms exist for sequential query evaluation: the semi-join algebra (Subsection 7.1), acyclic queries (Subsection 7.2), and worst-case optimal join evaluation (Subsection 7.3).

Related work. Due to space limitations, we only mention two other related papers. In a recent paper, query evaluation by circuits has been studied [28]. Although this is in principle closely related, the paper ignores polylogarithmic depth factors and therefore does not study $\mathcal{O}(1)$-time. The work of $\mathcal{O}(1)$-time PRAM algorithms has recently been studied in the context of dynamic complexity, where the database can change and the algorithms need to *maintain* the query result [25].

2 Preliminaries

In this section, we fix some notation and recall some concepts from database theory and PRAMs that are relevant for this paper. For a natural number n, we write $[n]$ for $\{1, \ldots, n\}$.

A *database schema* Σ is a finite set of relation symbols, where each symbol R is equipped with a finite set $\mathtt{attr}(R)$ of attributes. A tuple $t = (a_1, \ldots, a_{|X|})$ over a finite list $X = (A_1, \ldots, A_k)$ of attributes has, for each i, a value a_i for attribute A_i. Unless we are interested in the lexicographic order of a relation, induced by X, we can view X as a set. An R-relation is a finite set of tuples over $\texttt{attr}(R)$. The arity of R is $|\texttt{attr}(R)|$. For $Y \subseteq X$, we write $t[Y]$ for the restriction of t to Y. For $Y \subseteq \texttt{attr}(R)$, $R[Y] = \{t[Y] \mid t \in R\}$. A database D over Σ consists of an R-relation $D(R)$, for each $R \in \Sigma$. We usually write R instead of $D(R)$ if D is understood from the context. That is, we do not distinguish between relations and relation symbols. The size $|R|$ of a relation R is the number of tuples in R. By $|D|$ we denote the number of tuple entries in database D. For details on (the operators of) the relational algebra, we refer to [3]. We always assume a fixed schema and therefore a fixed maximal arity of tuples.

Parallel Random Access Machines (PRAMs). A *parallel random access machine* (PRAM) consists of a number of processors that work in parallel and use a shared memory. The memory is comprised of memory cells which can be accessed by a processor in $\mathcal{O}(1)$ time. Furthermore, we assume that the usual arithmetic and bitwise operations can be done in $\mathcal{O}(1)$ time by a processor. In particular, since the schema is considered fixed, a database tuple can be loaded, compared, etc. in $\mathcal{O}(1)$ time by one processor.

We mostly use the Concurrent-Read Concurrent-Write model (CRCW-PRAM), i.e. processors are allowed to read and write concurrently from and to the same memory location. More precisely, we mainly assume the *arbitrary* PRAM model: if multiple processors concurrently write to the same memory location, one of them, “arbitrarily”, succeeds. For some algorithms the *common* model would suffice, where all processors need to write the same value into the same location. We sometimes also use the weaker Exclusive-Read Exclusive-Write model (EREW-PRAM), where concurrent access is forbidden and the CREW-PRAM, where concurrent writing is forbidden. The work w of a PRAM computation is the sum of the number of all computation steps of all processors made during the computation. We define the space s required by a PRAM computation as the maximal index of any memory cell accessed during the computation. We refer to [18] for more details on PRAMs and to [26, Section 2.2.3] for a discussion of alternative space measures.

In principle, we assume that relations are stored as sequences of tuples, i.e., as arrays. Informally an array $\mathcal{A}$ is a sequence $t_1, \dots, t_N$ and it represents a relation R, if each tuple from R appears once as some t_i. In Section 4, we describe more precisely, how databases are represented for our PRAM algorithms.

3 Obstacles

We next discuss some obstacles that pose challenges for $\mathcal{O}(1)$-time parallel algorithms for query evaluation. They stem from various lower bound results from the literature.

The first obstacle, already mentioned in the introduction, is that we cannot expect that query results can be stored in arrays in a compact fashion, that is, as a sequence $t_1, \dots, t_m$ of tuples, for a result with m tuples. This follows from the following lower bound on the *linear approximate compaction problem*, where the, somewhat relaxed, goal is to move m scattered tuples into a target array of size $4m$.

▶ **Proposition 3.1** ([22, Theorem 4.1]). *Solving the linear approximate compaction problem on a randomized strong priority CRCW-PRAM requires $\Omega(\log^* n)$ expected time.*

Since the PRAM model of that bound is stronger than the arbitrary CRCW model, it applies to our setting.

The following theorem illustrates, how this lower bound restrains the ability to compute query results in a compact form, even if the input relations are given by compact arrays. Analogous results can be shown for simple projection and selection queries.

▶ **Theorem 3.2.** *Let q be the conjunctive query defined by $q : H(x) \leftarrow R(x), S(x)$. Every algorithm, which, upon input of arrays for relations R and S, computes an array of size $|q(D)|$ for $q(D)$ without empty cells, requires $\omega(1)$ time. This holds even if the arrays for R and S are compact and their entries are lexicographically ordered.*

Proof sketch. Towards a contradiction, we assume that there is an algorithm on a PRAM which computes in constant time, upon input of an input database D, an array of size $|q(D)|$ for the query result $q(D)$. This can be used to solve the linear approximate compaction problem in constant time as follows. Let $\mathcal{A}$ be an instance for the linear approximate compaction problem with m non-empty cells. In the first step create an array $\mathcal{A}_R$ of size n for the unary relation R that consists of even numbers 2 to $2n$. This can be done in parallel in constant time with n processors.

In the second step, store, for every $i \in \{1,\ldots,n\}$, the value $2i$ in the array $\mathcal{A}_S$ of size n for relation S if $\mathcal{A}[i]$ has a value (i.e. the i-th cell is not empty), and $2i+1$, otherwise. Hence, the query result of q exactly consists of even numbers $2i$ where i is an index such that $\mathcal{A}[i]$ has a value. From the array for the query result a solution for the linear approximate compaction problem can be obtained by replacing every value $2i$ in the result by $\mathcal{A}[i]$. The size of the compact array is m.

All in all, the algorithm takes constant time to solve the linear approximate compaction problem. This is a contradiction to Proposition 3.1. We note that by construction, the arrays for the relations R and S are ordered and have no empty cells. ◀

As a consequence of Theorem 3.2, our main data structure to represent relations are arrays that might contain *empty cells* that do not correspond to tuples in the relation.

As the example in the introduction illustrated, it is important that intermediate results can be compacted to some extent. Indeed, processors can be assigned to all cells of an array, but not so easily to only the non-empty cells. We extensively use a classical technique by Hagerup [14] that yields some (non-linear) compaction (see Proposition 5.1 for the statement of this result). However, Hagerup's compaction algorithm does not preserve the order of the elements in the array, so ordered arrays with non-empty cells are transformed into more compact but unordered arrays.

This brings us to another notorious obstacle for $\mathcal{O}(1)$-time parallel algorithms: they can not sort with polynomially many processors, even not in a (slightly) non-compact fashion. The *padded sort problem* asks to sort n given items into an array of length $n + o(n)$ with empty cells in spots without an item.

▶ **Proposition 3.3** ([22, Theorem 4.2]). *Solving the padded sort problem on a randomized strong priority CRCW-PRAM requires $\Omega(\log^* n)$ expected time.*

Thus, we cannot rely on sorting as an intermediate operator, either.

Yet another weakness of $\mathcal{O}(1)$-time parallel algorithms is counting. It follows readily from the equivalence with polynomial-size constant-depth circuits that (reasonable) CRCW-PRAMs cannot tell whether the number of ones in a sequence of zeros and ones is even [12, 2], let alone count them. These obstacles apply in particular to the evaluation of aggregate queries and for query evaluation under bag semantics with explicit representation of multiplicities of tuples. However, we do not study any of those in this paper.

Note. It turned out at submission time of this paper that we had missed a paper by Goldberg and Zwick [13], that contains two improvements to the above: it shows that *ordered* compaction and approximate counting, up to a factor of $1 + \frac{1}{(\log n)^a}$, for any $a > 0$, are possible in constant time with work $\mathcal{O}(n^{1+\varepsilon})$. In fact both results rely on the same technique for computing consistent approximate prefix sums. We discuss this issue further in our conclusion.

4 Basics

Query evaluation algorithms often use additional data structures like index structures. We consider two different kinds of such data structures for our $\mathcal{O}(1)$-time parallel algorithms.

The first setting that we consider is that of dictionary-based compressed databases, see, e.g., [10]. In a nutshell, the database has a dictionary that maps data values to natural numbers and internally stores and manipulates tuples over these numbers to improve performance. Such dictionaries are often defined attribute-wise, but for the purpose of this paper this does not matter. Query evaluation does not need to touch the actual dictionaries, it only works with the numbers. In this paper, we write "in the presence of a dictionary" or "in the dictionary setting" to indicate that we assume that such a dictionary exists for the database D at hand, and that it uses numbers of size at most $\mathcal{O}(|D|)$. In particular, the database relations then only contain numbers of this size.

In the other *ordered* setting, we assume that database relations are represented by ordered arrays, for each order of its attributes. In particular, we assume a linear order on the data values.

Arrays. As mentioned before, we assume in this paper that relations are stored in 1-dimensional arrays, whose entries are tuples that might be augmented by additional data.

More formally, an array $\mathcal{A}$ is a sequence of consecutive memory cells. The number of cells is its *size* $|\mathcal{A}|$. By $\mathcal{A}[i]$ for $1 \leq i \leq |\mathcal{A}|$ we refer to the i-th cell of $\mathcal{A}$ and to the (current) content of that cell, and we call i its index. We assume that the size of an array is always available to all processors (for instance, it might be stored in a "hidden" cell with index 0). Given an index i, any processor can access cell $\mathcal{A}[i]$ in $\mathcal{O}(1)$ time with $\mathcal{O}(1)$ work.

In this paper, a cell $\mathcal{A}[i]$ of an array always holds a distinguished database tuple $t = \mathcal{A}[i].t$ (over some schema) and a flag that indicates whether the cell is inhabited.[1] There might be additional data, e.g., further Boolean flags and links to other cells of (possibly) other arrays.

We say that an array *represents* a relation R if some inhabited cell holds tuple t, for each tuple t of R, and no inhabited cells contain other tuples. It represents R *concisely*, if each tuple occurs in exactly one inhabited cell. To indicate that an array represents a relation R we usually denote it as $\mathcal{A}_R$, $\mathcal{A}'_R$, etc.

An array $\mathcal{A}$ that represents a relation R is *ordered* if it is lexicographically ordered with respect to some ordered list X of the attributes from R's schema in the obvious sense. Its order is Y-compatible, for a set Y of attributes, if the attributes of Y form a prefix of X (hence the order induces a partial order with respect to Y).

We often consider the *induced tuple sequence* $t_1, \ldots, t_{|\mathcal{A}|}$ of an array $\mathcal{A}$. Here, $t_i = \mathcal{A}[i].t$ is a *proper* tuple, if $\mathcal{A}[i]$ is inhabited, or otherwise t_i is the *empty tuple* $\perp$.

▶ **Example 4.1.** The tuple sequence $[(1,5), \perp, (3,4), (8,3), (1,5), \perp, \perp, (7,3)]$ from an array $\mathcal{A}$ of size eight has five proper tuples and three empty tuples. It represents the relation $R = \{(1,5), (3,4), (8,3), (7,3)\}$, but *not* concisely. The sequence $[(1,5), (3,4), (7,3), \perp, (8,3)]$ represents R concisely and ordered with respect to the canonical attribute order.

Operations and links. Before we explain the basic operations used by our evaluation algorithms we illustrate some aspects by an example.

[1] If a cell is not inhabited, its data is basically ignored.

▶ **Example 4.2.** We sketch how to evaluate the projection $\pi_B(R)$ given the array $\mathcal{A} = [(1,5),(3,4),(7,3),\perp,(8,3)]$ from Example 4.1.

First, with the operation `Map` the array $\mathcal{A}'[5,4,3,\perp,3]$ is computed and each tuple $\mathcal{A}[i]$ is augmented by a link to $\mathcal{A}'[i]$ and vice versa. To achieve this, the tuples from $\mathcal{A}$ are loaded to processors $1,\dots,5$, each processor applies the necessary changes to its tuple, and then writes the new tuple to the new array $\mathcal{A}'$. We note that it is not known in advance which cells of $\mathcal{A}$ are inhabited and therefore, we need to assign one processor for each cell. Each processor only applies a constant number of steps, so the overall work for the `Map`-operation is $\mathcal{O}(|\mathcal{A}|)$.

To get an array that represents $\pi_B(R)$ concisely, a second operation eliminates duplicates. To this end, it creates a copy $\mathcal{A}''$ of $\mathcal{A}'$ and checks with one processor for each pair (i,j) of indices with $i < j$ in parallel, whether $\mathcal{A}''[i] = \mathcal{A}''[j]$ holds. If $\mathcal{A}''[i] = \mathcal{A}''[j]$ holds, then cell $\mathcal{A}''[i]$ is made uninhabited. Lastly, the algorithm creates links from every cell in $\mathcal{A}'$ to the unique inhabited cell in $\mathcal{A}''$ holding the same tuple. To do this, it checks with one processor for each pair (i,j) of indices with $i < j$ in parallel, whether $\mathcal{A}'[i] = \mathcal{A}''[j]$ holds and $\mathcal{A}''[j]$ is inhabited. If this is the case, the processor for (i,j) augments the cell $\mathcal{A}'[i]$ with a link to $\mathcal{A}''[j]$ and vice versa. Note that multiple processors might attempt to augment a cell $\mathcal{A}''[j]$ with a link to a cell of $\mathcal{A}'$; but since this happens in parallel only one processor will be successful. Overall, we get 2-step-links from $\mathcal{A}$ to $\mathcal{A}''$ and 2-step-links from $\mathcal{A}''$ to "representatives" in $\mathcal{A}$.

The second operation has a work bound of $\mathcal{O}(|\mathcal{A}|^2)$ because it suffices to assign one processor for each pair (i,j) of indices and each processor only applies a constant number of steps. We will show in Section 5 that work bounds $\mathcal{O}(|\mathcal{A}|)$ and $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ can be achieved for eliminating duplicates in the dictionary and ordered setting, respectively (cf. Lemma 5.10).

Of course, one might skip the intermediate writing and reading of tuples of $\mathcal{A}'$. We will often blur the distinction whether tuples reside in an array or within a sequence of processors.

Basic operations. Next, we describe some basic operations which we will use as building blocks in the remainder of this paper to query evaluation algorithms for PRAMs. We will describe algorithms for them in the next section.

Just as in Example 4.2, the operations usually get arrays as input, produce arrays as output, augment tuples and add links between tuples. In fact, each time a tuple of a new array results from some tuple of an input array we silently assume that (possibly mutual) links are added.

`Concatenate`$(\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}})$ computes a new array $\mathcal{C}$ of size $|\mathcal{A}| + |\mathcal{B}|$ consisting of the tuples from $\mathcal{A}$ followed by the tuples from $\mathcal{B}$ in the obvious way. As usual, mutual links are added, in particular, the link from $\mathcal{B}[1]$ indicates where the tuples from $\mathcal{B}$ start in $\mathcal{C}$.

`Map`$(\boldsymbol{\mathcal{A}}, \boldsymbol{f})$ returns an array $\mathcal{B}$ with $\mathcal{B}[i] = f(\mathcal{A}[i])$, where f is a function that maps proper tuples in $\mathcal{A}$ to empty or proper (possibly augmented) tuples and empty to empty tuples.

`Partition`$(\boldsymbol{\mathcal{A}}, \boldsymbol{n}, \boldsymbol{g})$ yields n arrays $\mathcal{A}_1,\dots,\mathcal{A}_n$ of size $|\mathcal{A}|$ and adds links. Here, g is a function that maps proper tuples in $\mathcal{A}$ to numbers in $[n]$. For every $i \in [|\mathcal{A}|]$ and every $j \in [n]$, $\mathcal{A}_j[i] = \mathcal{A}[i]$, if $\mathcal{A}[i]$ is inhabited and $g(\mathcal{A}[i]) = j$, otherwise $\mathcal{A}_j[i]$ is uninhabited.

`Compact`$_\varepsilon(\boldsymbol{\mathcal{A}})$ copies the proper tuples in $\mathcal{A}$ into distinct cells of an array $\mathcal{B}$ of size at most $b^{i\varepsilon}$, where $i \le \frac{1}{\varepsilon}(1+\varepsilon)$ is a non-negative integer[2] such that $b^{i\varepsilon} \le n^{1+\varepsilon} b^{\varepsilon}$, and b is $|\mathcal{A}|$ or an upper bound for the (possibly unknown) number n of proper tuples in $\mathcal{A}$, given as an optional parameter. We refer to i as the "compaction parameter" and note that it can be inferred from the size of $\mathcal{B}$. Mutual links are added as usual.

[2] In the special case $n = 0$, $i = 0$ is required.

`SearchRepresentatives`($\mathcal{A}, \mathcal{B}$) links every inhabited cell $\mathcal{A}[i]$ to an inhabited representative cell $\mathcal{B}[j]$, such that $\mathcal{B}[j].t = \mathcal{A}[i].t$ holds, if such a cell exists. Furthermore, for every i_1, i_2 with $\mathcal{A}[i_1] = \mathcal{A}[i_2] \neq \bot$, both $\mathcal{A}[i_1]$ and $\mathcal{A}[i_2]$ are linked to the same representative. If required a copy of $\mathcal{B}$ might be produced in which representatives are marked. We stress that $\mathcal{B}$ does not have to represent its relation concisely, and, in fact, the operation is used to remove duplicates.

`Deduplicate`($\mathcal{A}$) chooses one representative tuple for each tuple-value, marks the remaining cells as uninhabited and redirects incoming links from other arrays towards the representatives, if possible.

5 Algorithmic Techniques and Algorithms for Basic Array Operations

In this section, we first describe some important algorithmic techniques and present algorithms for the basic operations established in Section 4, afterwards.

Compaction. To implement the operation $\texttt{Compact}_\varepsilon$, we will utilise the following classical result by Hagerup, whose formulation is slightly adapted to our setting.

▶ **Proposition 5.1** ([14], Unnumbered theorem, p. 340)**.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given an array $\mathcal{A}$ and a number k, copies the proper tuples in $\mathcal{A}$ to distinct cells of an array of size at most $k^{1+\varepsilon}$ or detects that $\mathcal{A}$ contains more than k proper tuples. The algorithm requires $\mathcal{O}(|\mathcal{A}|)$ work and space on an arbitrary CRCW-PRAM.*

The space bound is only implicit in [14]. For the sake of convenience, we give a detailed account of the algorithm in the full version of this paper [19], including an analysis of the space requirements.

Array hash tables. In the presence of dictionaries we use *array hash tables* which associate each inhabited cell in $\mathcal{A}$ with a number from $[|\mathcal{A}|]$, such that $\mathcal{A}[i], \mathcal{A}[j]$ get the same number if and only if $\mathcal{A}[i].t = \mathcal{A}[j].t$ holds. Array hash tables can be efficiently computed.

▶ **Lemma 5.2.** *There is a $\mathcal{O}(1)$-time parallel algorithm that, in the presence of a dictionary, computes an array hash table for a given array $\mathcal{A}$, and requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space on an arbitrary CRCW-PRAM.*

We note that due to the "arbitrary" resolution of concurrent write, the result of such a computation is not uniquely determined by the relations.

Proof sketch. Let $A_1, \dots, A_\ell$ be the attributes of the relation R represented by $\mathcal{A}$ in an arbitrary but fixed order and define $X_j = \{A_1, \dots, A_j\}$ for all $j \in \{1, \dots, \ell\}$. The algorithm inductively computes hash values for tuples in $R[X_j]$ for increasing j from 1 to ℓ.

The idea is to assign, to each tuple $t \in R[X_j]$, a processor number in the range $\{1, \dots, |\mathcal{A}|\}$ as hash value and augment each cell of $\mathcal{A}$ containing a proper tuple t' with $t'[X] = t$ by this hash value. Since the same (projected) tuple $t \in R[X_j]$ might occur in multiple, pairwise different, cells of $\mathcal{A}$, it does not suffice to load all tuples in $\mathcal{A}$ to $|\mathcal{A}|$ processors and let each processor augment the tuple loaded to it by its processor number: multiple (different) numbers might get assigned to the same tuple (in different cells of $\mathcal{A}$). To resolve these conflicts, the algorithm utilizes the presence of a dictionary and the hash values for tuples in $R[X_{j-1}]$.

For the base case $j = 1$ the algorithm allocates an auxiliary array of size $\mathcal{O}(|D|)$ and loads the tuples in $\mathcal{A}$ to processors 1 to $|\mathcal{A}|$. To be more precise, for each $i \in \{1, \dots, |\mathcal{A}|\}$, the tuple t_i in cell $\mathcal{A}[i]$ is loaded to processor i. Recall that, in the dictionary setting, each value in the active domain is a number of size at most $\mathcal{O}(|D|)$. Thus, the projection $t[A_1]$ can be used as index for the auxiliary array. Each processor i with a proper tuple writes its processor number i into cell $t_i[A_1]$ of the auxiliary array and then assigns to t_i the value actually written at position $t_i[A_1]$. Note that, for each value a, all processors i with $t_i[A_1] = a$ will assign the same value to their tuple t_i, since precisely one processor among the processors with $t_i[A_1] = a$ succeeds in writing its number to cell $t_i[A_1]$ on an arbitrary CRCW-PRAM. This can be done with $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|D|)$ space.

For $j > 1$ the algorithm proceeds similarly but also takes, for a tuple t, the hash value $h_{j-1}(t[X_{j-1}])$ for $t[X_{j-1}]$ into account, in addition to $t[A_j]$. For this purpose, the algorithm first computes the hash values for $R[X_{j-1}]$. It then allocates an auxiliary array of size $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ which is interpreted as two-dimensional array and each processor i writes its number into cell $(h_{j-1}(t_i[X_{j-1}]), t_i[X_j])$ of the auxiliary array (if t_i is a proper tuple). The number in this cell is then the hash value for $t_i[X_j]$.

Writing and reading back the processor numbers requires $\mathcal{O}(|\mathcal{A}|)$ work and the auxiliary array requires $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space. The same bounds hold for the recursive invocations of `ComputeHashvalues`. Since the recursion depth is ℓ, the procedure requires $\mathcal{O}(\ell \cdot |\mathcal{A}|) = \mathcal{O}(|\mathcal{A}|)$ work and, because the space for the auxiliary arrays can be reused, $\mathcal{O}(|\mathcal{A}| \cdot |D|)$ space in total. ◀

Since most of our query evaluation algorithms rely on these two techniques, we adopt the arbitrary CRCW-PRAM as our standard model and refer to it simply by *CRCW-PRAM*.

Search in ordered arrays. In sequential database processing, indexes implemented by search trees play an important role, in particularly for the test whether a given tuple is in a given relation. We use ordered arrays instead. Our search algorithm for ordered arrays uses links from each cell to the next and previous inhabited cell. We refer to those links as predecessor and successor links, respectively, and say an array is *fully linked* if it has predecessor and successor links.

▶ **Proposition 5.3.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for an array $\mathcal{A}$, predecessor and successor links with work $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ on a common CRCW-PRAM.*

Proof sketch. We describe the computation of predecessor links. Successor links can be computed analogously. Let $n = |\mathcal{A}|$ and $\delta = \frac{\varepsilon}{2}$. In the first round, the algorithm considers subintervals of length n^δ and establishes predecessor links within them. To this end, it uses, for each interval a $n^\delta \times n^\delta$-table whose entries (i, j) with $i < j$ are initialised by 1, if $\mathcal{A}[i]$ is inhabited and, otherwise, by 0. Next, for each triple i, j, k of positions in the interval entry (i, j) is set to 0 if $i < k < j$ and $\mathcal{A}[k]$ is inhabited. It is easy to see that afterwards entry (i, j) still carries a 1 if and only if i is the predecessor of j. And for all such pairs a link from $\mathcal{A}[j]$ to $\mathcal{A}[i]$ is added. For every interval, $(n^\delta)^3 = n^{3\delta}$ processors suffice for this computation, i.e. one processor for each triple i, j, k of positions in the interval. Since there are $\frac{n}{n^\delta} = n^{1-\delta}$ intervals, this yields an overall work of $n^{1-\delta} \cdot n^{3\delta} = n^{1+2\delta} = n^{1+\varepsilon}$. In the next round, intervals of length $n^{2\delta}$ are considered and each is viewed as a sequence of n^δ smaller intervals of length n^δ. The goal in the second round is to establish predecessor links for the minimum cells of each of the smaller intervals. This can be done similarly with the same asymptotic work as round 1. After $\lceil \frac{1}{\delta} \rceil$ rounds, this process has established predecessor links for all cells (besides for the minimum cells without a predecessor). ◀

In fully linked ordered arrays, tuples can be searched for efficiently.

▶ **Proposition 5.4.** *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that computes, for a given tuple t and an ordered array $\mathcal{A}$ with predecessor and successor links, the largest tuple t' in $\mathcal{A}$ with $t' \le t$ with work $\mathcal{O}(|\mathcal{A}|^{\varepsilon})$ on an CREW-PRAM.*

Proof sketch. Let $n = |\mathcal{A}|$. In the first round, using n^{ε} processors, the algorithm tests for all cells with positions $k = in^{1-\varepsilon}$ whether $\mathcal{A}[k]$ is inhabited, or the predecessor of $\mathcal{A}[k]$ contains a tuple $t' \le t$ and whether this does not hold for position $(i+1)n^{1-\varepsilon}$ or its successor. By a suitable process the search continues recursively in the thus identified sub-interval. After $\lceil \frac{1}{\varepsilon} \rceil$ rounds it terminates. Since, in each round, n^{ε} processors are used, the statement follows. ◀

We note that analogously it is possible to search for m tuples in parallel with work $\mathcal{O}(m\,|\mathcal{A}|^{\varepsilon})$.

As an alternative to ordered arrays, bounded-depth search trees could be used. They can be defined in the obvious way with degree about n^{ε}. The work for a search is asymptotically the same as for fully linked ordered arrays.

Algorithms for basic array operations. In the remainder of this section we consider algorithms for basic array operations. For the operations `Concatenate`, `Map`, and `Partition`, neither the algorithm nor the analysis depends on the setting, i.e. they are the same in the dictionary setting and the ordered setting. Furthermore, their implementation is straightforward and only requires EREW-PRAMs, instead of CRCW-PRAMs.

▶ **Lemma 5.5.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Concatenate` *that, given arrays $\mathcal{A}$ and $\mathcal{B}$, requires $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ work and space on an EREW-PRAM.*

Proof sketch. The tuples in $\mathcal{A}$ are loaded to processors 1 to $|\mathcal{A}|$ and the tuples in $\mathcal{B}$ to processors $|\mathcal{A}| + 1$ to $|\mathcal{A}| + |\mathcal{B}|$. The tuples are then stored in the output array $\mathcal{C}$. Each processor can also augment its tuple (between $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$) with mutual links. ◀

▶ **Lemma 5.6.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Map` *that, given an array $\mathcal{A}$ and a function f that can be evaluated in $\mathcal{O}(1)$-time with work and space $\mathcal{O}(1)$ on an EREW-PRAM, requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}|)$ space on an EREW-PRAM. If f is order-preserving and $\mathcal{A}$ is ordered, then the output array is ordered, too.*

Proof sketch. The algorithm loads the tuples in $\mathcal{A}$ to processors 1 to $|\mathcal{A}|$ and each processor computes the image $f(t)$ for its tuple t.

Since $f(t)$ has to be computed for $|\mathcal{A}|$ tuples, the bounds for work and space follow. ◀

▶ **Lemma 5.7.** *There is a $\mathcal{O}(1)$-time parallel algorithm for* `Partition` *that, given an array $\mathcal{A}$, an integer n, and a function g that maps proper tuples in $\mathcal{A}$ into $\{1,\dots,n\}$ and can be evaluated in $\mathcal{O}(1)$-time with work and space $\mathcal{O}(1)$, requires $\mathcal{O}(n \cdot |\mathcal{A}|)$ work and $\mathcal{O}(n \cdot |\mathcal{A}|)$ space on an EREW-PRAM.*

Proof sketch. The algorithm first augments every proper tuple t with the number $g(t)$ using `Map`. This requires $\mathcal{O}(|\mathcal{A}|)$ work and $\mathcal{O}(|\mathcal{A}|)$ space, cf. Lemma 5.6.

Then the arrays $\mathcal{A}_1, \dots, \mathcal{A}_n$ of size $|\mathcal{A}|$ are allocated (and initialised). This requires $\mathcal{O}(n \cdot |\mathcal{A}|)$ work and space.

For each $i \in \{1,\dots,|\mathcal{A}|\}$ in parallel, the tuple t_i in cell $\mathcal{A}[i]$ is then – if it is a proper tuple – copied into cell $\mathcal{A}_j[i]$ where $j = g(t_i)$. This requires $\mathcal{O}(\mathcal{A})$ work. ◀

Let us point out that the upper bound for the work stated in Lemma 5.7 can be reduced to $\mathcal{O}(n + |\mathcal{A}|)$ by adapting the classical *lazy array initialisation technique* for (sequential) RAMs to PRAMs.[3] It turned out, however, that this is not necessary for our results, since n is either a constant or the work is dominated by other operations in our algorithms.

The algorithm for $\texttt{Compact}_\varepsilon$ does not depend on the setting either. It is implicitly proved in the proof of Proposition 5.1 in [14] for a fixed choice of b. The idea is to try several, but constantly many, compaction parameters.

▶ **Lemma 5.8** ([14], implicit in proof). *For every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm for $\texttt{Compact}_\varepsilon$ that, given an array $\mathcal{A}$ and an upper bound b for the number of proper tuples in $\mathcal{A}$, requires $\mathcal{O}(|\mathcal{A}|)$ work and space on a CRCW-PRAM.*

Proof sketch. Let n denote the number of proper tuples in the given array $\mathcal{A}$.

We assume $n \geq 1$ in the following[4] and set $k = \lceil \frac{1}{\varepsilon} \rceil$. The algorithm invokes the algorithm guaranteed by Proposition 5.1 with $k_i = b^{\frac{i\varepsilon}{1+\varepsilon}}$ for every $0 \leq i \leq \lceil k(1+\varepsilon) \rceil$ in ascending order until it is successful (or the current k_i is larger than $|\mathcal{A}|$ in which case the procedure can just return $\mathcal{A}$). Each of the (constantly many) invocations requires $\mathcal{O}(|\mathcal{A}|)$ work and space.

If the algorithm is successful for $i = 0$, the requirements are trivially met.

Otherwise, let i be such that the compaction for $i+1$ was successful but the compaction for i was not (note that for $j = \lceil k(1+\varepsilon) \rceil$ the compaction will always succeed, since k_j is an upper bound for the number of proper tuples in $\mathcal{A}$). Then the resulting array has size

$$k_{i+1}^{1+\varepsilon} = \left(b^{\frac{(i+1)\varepsilon}{1+\varepsilon}}\right)^{1+\varepsilon} = b^{(i+1)\varepsilon} = b^{i\varepsilon} \cdot b^{\varepsilon}.$$

Moreover, since the compaction algorithm was not successful for i, we also have that $k_i = b^{\frac{i\varepsilon}{1+\varepsilon}} < n$ and, thus, $b^{i\varepsilon} < n^{1+\varepsilon}$.

All in all, the array $\mathcal{B}$ has size $b^{(i+1)\varepsilon} \leq n^{1+\varepsilon} \cdot b^{\varepsilon}$. ◀

For $\texttt{SearchRepresentatives}(\mathcal{A}, \mathcal{B})$ we present four algorithms, depending on the setting and, in the ordered setting, whether $\mathcal{A}$ or $\mathcal{B}$ is suitably ordered. This operation is crucial for deduplication, semi-join and join and the upper bounds impact the bounds for those operations as well.

▶ **Lemma 5.9.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* $\texttt{SearchRepresentatives}$ *that, given arrays $\mathcal{A}$ and $\mathcal{B}$, have the following bounds on a CRCW-PRAM.*

(a) *Work $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|) \cdot |\mathcal{B}|)$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, without any assumptions;*

(b) *Work $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|) \cdot |D|)$ in the presence of a dictionary;*

(c) *Work $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, if $\mathcal{B}$ is ordered and fully linked;*

(d) *Work $\mathcal{O}(|\mathcal{B}| \cdot |\mathcal{A}|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|))$, if $\mathcal{A}$ is ordered and fully linked and $\mathcal{B}$ is concise.*

Proof sketch. For (a), the naive algorithm can be used. In a first phase, it uses one processor per pair (i, j) of indices for the cells of $\mathcal{B}$ to mark duplicates in $\mathcal{B}$: if $i < j$ and $\mathcal{B}[i].t = \mathcal{B}[j].t$, then $\mathcal{B}[j]$ is marked as duplicate. On the second phase, it uses one processor per pair (i, j) of indices for the cells of $\mathcal{A}$ and $\mathcal{B}$ and links $\mathcal{A}[i]$ to $\mathcal{B}[j]$ if $\mathcal{B}[i].t = \mathcal{B}[j].t$ and $\mathcal{B}[j]$ is not marked as duplicate.

[3] In a nutshell, this requires replacing a global counter by one counter per processor and maintaining back-references to initialised cells per processor (processors can still read counters and back-references of other processors).

[4] The algorithm yields $i = 0$, if $n = 0$ holds, as required.

For (b), first an array hash table for (the concatenation of) $\mathcal{A}$ and $\mathcal{B}$ is computed with $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ work and $\mathcal{O}((|\mathcal{A}|+|\mathcal{B}|)\cdot|D|)$ space, thanks to Lemma 5.2 and Lemma 5.5. For a proper tuple t, let $h(t)$ denote the hash value in the range $\{1,\ldots,|\mathcal{A}|+|\mathcal{B}|\}$ assigned to t. The algorithm then allocates an auxiliary array of size $|\mathcal{A}|+|\mathcal{B}|$ and, for each proper tuple s_i in $\mathcal{B}$, it writes, in parallel, i into cell $h(s_i)$ of the auxiliary array. Here s_i denotes the i-tuple from $\mathcal{B}$. Other processors might attempt to write an index to cell $h(s_i)$ but only one will succeed. This requires $\mathcal{O}(|\mathcal{B}|)$ work to write the indices and $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ space for the auxiliary array.

For each proper tuple t in $\mathcal{A}$ it is then checked in parallel, if cell $h(t)$ contains an index i. If yes, then t is marked and augmented with a pointer to cell $\mathcal{B}[i]$, since $\mathcal{B}[i].t = t$. If not, then t has no partner tuple in $\mathcal{B}$, thus t is not augmented by a link.

Towards (c), the algorithm identifies, for each proper tuple t in $\mathcal{A}$, the smallest proper tuple s in $\mathcal{B}$ such that $t \leq s$. If $t = s$ holds, t's cell is marked and a link to the cell of s is added. For each tuple, this can be done with work $|\mathcal{B}|^{\varepsilon}$, thanks to Proposition 5.4 and these searches can be done in parallel by assigning $|\mathcal{B}|^{\varepsilon}$ processors per tuple of $\mathcal{A}$.

For (d), the algorithm searches, for each proper tuple s in $\mathcal{B}$, the smallest tuple t in $\mathcal{A}$ with $t \geq s$. If $t = s$ then the cell of t is marked and a link to the cell of s is added. If $\mathcal{A}$ is guaranteed to be concise, that's all. Otherwise, for each proper tuple t in $\mathcal{A}$ the smallest inhabited cell $\mathcal{A}[i]$ with $\mathcal{A}[i].t = t$ is searched. If it is marked then t is marked as well and a link to the cell in $\mathcal{B}$ to which $\mathcal{A}[i]$ links is added. ◀

▶ **Lemma 5.10.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for* `Deduplicate` *that, given an array $\mathcal{A}$ have the following bounds on an arbitrary CRCW-PRAM.*

(a) *Work $\mathcal{O}(|\mathcal{A}|^2)$ and space $\mathcal{O}(|\mathcal{A}|)$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}|)$ and space $\mathcal{O}(|\mathcal{A}|\cdot|D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}|^{1+\varepsilon})$ and space $\mathcal{O}(|\mathcal{A}|)$, if $\mathcal{A}$ is ordered.*

Proof sketch. In all three cases, `SearchRepresentatives`$(\mathcal{A},\mathcal{A})$ is invoked and afterwards all inhabited cells with tuples that received a link to a different tuple are made uninhabited, leaving only the other tuples as representatives. For (c), it might be necessary to compute full links according to Proposition 5.3. The bounds then follow with Lemma 5.9. ◀

6 Algorithms for Database Operations

In this section, we present $\mathcal{O}(1)$-time parallel algorithms for the operators of the relational algebra and analyse their complexity with respect to work and space.

We formulate the results for relations rather than arrays. We always assume that a relation R is represented concisely by an array $\mathcal{A}_R$, but we make no assumptions about the compactness of the representation. All algorithms produce output arrays which represent the result relation concisely.

With the notable exception of the join operator, for most operators the algorithms are simple combinations of the algorithms of Section 5. The respective proofs are given in the full version of this paper [19]. The algorithms for the join operator are more involved and are presented at the end of the section.

We note that the relational algebra has an additional rename operator, which, of course, does not require a parallel algorithm.

▶ **Proposition 6.1.** *There is a $\mathcal{O}(1)$-time parallel algorithm that receives as input a relation R and an attribute X in R, and an element a in the domain or an attribute Y, and computes the selection $\sigma_{X=a}(R)$ (if a is given) or $\sigma_{X=Y}(R)$ (if Y is given). The algorithm requires $\mathcal{O}(|\mathcal{A}_R|)$ work and space on an EREW-PRAM. The output array is of size at most $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too.*

The algorithm is a simple application of the operation `Map`.

▶ **Proposition 6.2.** *For every $\varepsilon > 0$, there are $\mathcal{O}(1)$-time parallel algorithms for CRCW-PRAMs that compute upon input of two relations R and S the semijoin $R \ltimes S$ with the following bounds. Here, X denotes the joint attributes of R and S.*
(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is X-compatibly ordered and fully linked;*
(d) *Work $\mathcal{O}(|\mathcal{A}_S| \cdot |\mathcal{A}_R|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_R$ is X-compatibly ordered and fully linked and $\mathcal{A}_S$ is concise.*
The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too. Moreover, each $t \in R \ltimes S$ in the output of $R \ltimes S$ gets augmented by a link to a corresponding tuple in S.

▶ **Proposition 6.3.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the difference $R \setminus S$ with the following bounds.*
(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is ordered and fully linked;*
(d) *Work $\mathcal{O}(|\mathcal{A}_S| \cdot |\mathcal{A}_R|^{\varepsilon})$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_R$ is ordered and fully linked and $\mathcal{A}_S$ is concise.*
The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered, then the output is ordered, too.

The algorithms for Proposition 6.2 and Proposition 6.3 combine the appropriate algorithm for `SearchRepresentatives` with suitable applications of `Map`.

▶ **Proposition 6.4.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that receive as input a relation R and a list X of attributes from R, and evaluate the projection $\pi_X(R)$ with the following bounds.*
(a) *Work $\mathcal{O}((|\mathcal{A}_R|^2)$ and space $\mathcal{O}(|\mathcal{A}_R|)$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R|)$ and space $\mathcal{O}(|\mathcal{A}_R| \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R|^{1+\varepsilon})$ and space $\mathcal{O}(|\mathcal{A}_R|)$, if $\mathcal{A}_R$ is X-compatibly ordered.*
The output array is of size $|\mathcal{A}_R|$. If $\mathcal{A}_R$ is ordered then the output is ordered, too.

The algorithms combine `Deduplicate` with `Map` in a straightforward manner. We note that we do not require in (c) that $\mathcal{A}_R$ is fully linked, since the work bound allows to compute links.

▶ **Proposition 6.5.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the union $R \cup S$ with the following bounds.*
(a) *Work $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, without any assumptions;*
(b) *Work $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|) \cdot |D|)$ in the presence of a dictionary;*
(c) *Work $\mathcal{O}(|\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon} + |\mathcal{A}_S|)$ and space $\mathcal{O}((|\mathcal{A}_R| + |\mathcal{A}_S|))$, if $\mathcal{A}_S$ is ordered and fully linked.*
The output array is of size $|\mathcal{A}_R| + |\mathcal{A}_S|$.

The algorithms basically concatenate $R \setminus S$ and S. We note that thanks to the symmetry of union, the algorithm of (c) can also be applied if $\mathcal{A}_R$ is ordered.

▶ **Proposition 6.6.** *For every $\varepsilon > 0$, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of two relations R and S the join $R \bowtie S$ with the following bounds. Here, X denotes the joint attributes of R and S.*

(a) *Work* $\mathcal{O}((|\mathcal{A}_S|^2 + |\pi_X(S)|^{1+\varepsilon} |\mathcal{A}_S|^{1+\varepsilon}) + (|\mathcal{A}_R| + |\mathcal{A}_S|) |\mathcal{A}_S| + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon} |\mathcal{A}_S|^{1+\varepsilon} + |\mathcal{A}_R| + |\mathcal{A}_S| + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$ *without any assumptions;*

(b) *Work* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon} |\mathcal{A}_S|^{1+\varepsilon} + (|\mathcal{A}_R| + |\mathcal{A}_S|) + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\pi_X(S)|^{1+\varepsilon} |\mathcal{A}_S|^{1+\varepsilon} + (|\mathcal{A}_R| + |\mathcal{A}_S|)|D| + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$ *in the presence of a dictionary;*

(c) *Work* $\mathcal{O}(|\mathcal{A}_S|^{1+\varepsilon} + |\mathcal{A}_R| \cdot |\mathcal{A}_S|^{\varepsilon} + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$ *and space* $\mathcal{O}(|\mathcal{A}_R| + |\mathcal{A}_S| + |R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon})$, *if* $\mathcal{A}_S$ *is* X*-compatibly ordered and fully linked.*

The output array is of size $|R \bowtie S| |\mathcal{A}_R|^{2\varepsilon} |\mathcal{A}_S|^{2\varepsilon}$.

Proof idea. The algorithms proceed in three phases, the grouping phase, the pairing phase and the joining phase. For (a) and (b), the tuples of S are grouped with respect to their X-attributes in the grouping phase. Each group is compacted into an array of some size $|\mathcal{A}_S|^{\ell\varepsilon}$. Likewise the projection $\pi_X(S)$, containing the "index tuples", is compacted. In the pairing phase, a semijoin reduction is performed and the remaining R-tuples are partitioned with respect to the size of their corresponding "X-group" from S. Finally, during the joining phase, output tuples are produced, by combining tuples from R with the tuples from their "X-group" from S. The work bounds for the three phases can be seen as the three main summands in the statement of the proposition.

If S is represented by an array that is X-compatibly ordered and fully linked, the grouping phase can be performed more efficiently. In that case, $\mathcal{A}_S$ itself can be viewed as the concatenation of all "X-groups". Thus, this steps is for free and, furthermore, the compaction of the "X-groups" can be done in-place and therefore only requires work $|\mathcal{A}_S|$ in total. The pairing phase and the joining phase are basically as for (a) and (b), but the work bounds for the pairing phase differ, due to the more efficient semijoin algorithm in the ordered setting. ◀

7 Query Evaluation

After studying algorithms for basic operations and operators of the relational algebra, we are now prepared to investigate the complexity of $\mathcal{O}(1)$-time parallel algorithms for query evaluation.

Although every query of the relational algebra can be evaluated by a $\mathcal{O}(1)$-time parallel algorithms with polynomial work, the polynomials can be arbitrarily bad. In fact, that a graph has a k-clique can be expressed by a conjunctive query with k variables and it follows from Rossman's $\omega(n^{k/4})$ lower bound for the size of bounded-depth circuit families for k-Clique [24] that any $\mathcal{O}(1)$-time parallel algorithm that evaluates this query needs work $\omega(n^{k/4})$.

We therefore concentrate in this section on restricted query evaluation settings. We study two restrictions of query languages which allow efficient sequential algorithms, the semijoin algebra and free-connex and/or acylic conjunctive queries. Furthermore, we present a $\mathcal{O}(1)$-time parallel version of worst-case optimal join algorithms.

In the following, IN always denotes the maximum number of tuples in any relation of the underlying database that is addressed by the given query. Furthermore, we always assume that the database relations are represented concisely by *compact* arrays without any uninhabited cells.

7.1 Semi-Join Algebra

The semijoin algebra is the fragment of the relational algebra that uses only selection, projection, rename, union, set difference and, not least, semijoin. It is well-known that semijoin queries produce only query results of size $\mathcal{O}(|D|)$ and can be evaluated in time $\mathcal{O}(|D|)$ [21, Theorem 7]. From the results of Section 6 we can easily conclude the following.

▶ **Proposition 7.1.** *For each query q of the semijoin algebra and for every $\varepsilon > 0$ there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that, given a database D, evaluate $q(D)$ with the following bounds.*
(a) *Work $\mathcal{O}(\mathsf{IN}^{2+\varepsilon})$ and space $\mathcal{O}(\mathsf{IN}^{2+\varepsilon})$, without any assumptions;*
(b) *Work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN} \cdot |D|)$ in the presence of a dictionary.*

Proof sketch. Towards (a), the operators of the query are evaluated with the naive algorithms from Section 6 (stated as (a)). After each evaluation the result array is compacted by $\texttt{Compact}_{\varepsilon/2}$. Statement (b) follows by using the (b)-algorithms from Section 6. ◀

Altogether, semijoin queries can be evaluated work-optimally by a $\mathcal{O}(1)$-time parallel algorithm. We plan to address ordered setting in a journal version of this paper. We expect that the results of [13] enable almost work-optimal $\mathcal{O}(1)$-time parallel algorithms with a $\mathcal{O}(\mathsf{IN}^{1+\varepsilon})$ work bound, if the relations are represented by suitably ordered arrays. We discuss this further in our conclusion.

7.2 Evaluation of Conjunctive Queries

In this section we give algorithms to evaluate subclasses of conjunctive queries in parallel. More precisely, we consider acyclic join queries, acyclic conjunctive queries, free-connex acyclic conjuntive queries and arbitrary free-connex conjunctive queries.

Conjunctive queries are conjunctions of relation atoms. We write a *conjunctive query* (*CQ* for short) q as a rule of the form $q : \mathsf{A} \leftarrow \mathsf{A}_1, \dots, \mathsf{A}_m$, where $\mathsf{A}, \mathsf{A}_1, \dots, \mathsf{A}_m$ are atoms and $m \geq 1$. A conjunctive query q is *acyclic*, if it has a join tree T_q, i.e. an undirected tree $(V(T), E(T))$ where $V(T)$ consists of the atoms in q and for each variable v in T_q the set $\{\alpha \in V(T) | \alpha \text{ contains } v\}$ induces a connected subtree of T_q. It is *free-connex acyclic* if q is acyclic and the Boolean query whose body consists of the body atoms *and* the head atom of q is acyclic as well [5, 9]. A *join query* is a conjunctive query with no quantified variable, i.e. every variable in a join query is free. For more background on (acyclic) conjunctive queries we refer to [1, 3].

Our algorithms rely on the well-known Yannakakis algorithm [29]. Yannakakis' algorithm receives as input an acyclic conjunctive query q, the join tree T_q and a database D. With each node v in T_q a relation S_v is associated. Initially, $S_v = R_v(D)$, where R_v is the relation that is labelled in v. The algorithm is divided into three steps.

(1) **bottom-up semijoin reduction:** All nodes are visited in bottom-up traversal order of T. When a node v is visited, S_v is updated to $S_v \ltimes S_c$ for every child c of v in T.
(2) **top-down semijoin reduction:** All nodes are visited in top-down traversal order of T. When a node v is visited, the relation S_c is updated to $S_c \ltimes S_v$ for every child c of v in T.
(3) All nodes are visited in bottom-up traversal order in T. When a node v is visited, the algorithm updates, for every child c of v, the relation S_v to $\pi_{\mathsf{free}(q) \cup \mathtt{attr}(S_v)}(S_v \bowtie S_c)$, where $\mathsf{free}(q)$ denotes the attributes that are associated with the free variables of q.

Proposition 6.2 immediately yields the following lemma.

▶ **Lemma 7.2.** *There are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms for phase (1) and (2) of the Yannakakis algorithm with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2)$ and space $\mathcal{O}(\mathsf{IN})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN} \cdot |D|)$ in the presence of a dictionary.*

By combining Yannakakis' algorithm with the algorithms from Section 6 we obtain the following results.

▶ **Proposition 7.3.** *For every $\varepsilon > 0$ and every acyclic join query q, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute $q(D)$, given a database D, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN}^{1+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$ and space $\mathcal{O}((\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon} |D|)$, in the presence of a dictionary.*

To perform phase (3) of the Yannakakis algorithm the parallel algorithms first shrink every array $\mathcal{A}_{R_v}$ to the size $|S_v|^{1+\varepsilon'} |R_v(D)|^{\varepsilon'}$ using $\texttt{Compact}_{\varepsilon'}(S_v)$, for some very small ε', depending (only) on the size of the join tree. Likewise, by calling the join algorithm with a suitable parameter, it strongly compacts each intermediate join result. That the stated bounds are met can be established by a straightforward, but tedious calculation, given in the full version of this paper [19].

▶ **Proposition 7.4.** *For every $\varepsilon > 0$, and every acyclic conjunctive query q, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute $q(D)$, given a database D, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{2+\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{IN}^{2+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{OUT}^{1+\varepsilon}\mathsf{IN}^{2+2\varepsilon})$ and space $\mathcal{O}(\mathsf{OUT}^{1+\varepsilon}\mathsf{IN}^{2+\varepsilon} |D|)$, in the presence of a dictionary.*

The algorithms are obtained from the algorithms for Proposition 7.3 by a suitable adaptation of phase (3). A proof sketch for Proposition 7.4 is given in the full version [19].

It turns out that the bounds for acyclic join queries carry over to free-connex acyclic conjunctive queries. We use the reduction from free-connex acyclic queries to join queries given in [8]. We adapt it for $\mathcal{O}(1)$-time parallel algorithms.

▶ **Lemma 7.5.** *For every free-connex acyclic query q and every database D there exists an acyclic join query $\tilde{q}$ and a database $\widetilde{D}$ such that $q(D) = \widetilde{q}(\widetilde{D})$. Here, $\tilde{q}$ only depends on q.*

Furthermore, there are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that compute upon input of a free-connex acyclic query q and a database D the corresponding join query $\widetilde{q}$ and database $\widetilde{D}$ with the following bounds.

(a) *Work $\mathcal{O}(\mathsf{IN}^2)$ and space $\mathcal{O}(\mathsf{IN})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN})$ and space $\mathcal{O}(\mathsf{IN} \cdot |D|)$ in the presence of a dictionary;*

A proof sketch for Lemma 7.5 is given in the full version of this paper [19]. By combining Lemma 7.5 and Proposition 7.3 we obtain the following result.

▶ **Corollary 7.6.** *There are CRCW-PRAM $\mathcal{O}(1)$-time parallel algorithms that receives as input a free-connex acyclic conjunctive query q and a database D and computes the result $q(D)$, with the following bounds.*

(a) *Work $\mathcal{O}(\mathsf{IN}^2 + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$ and space $\mathcal{O}(\mathsf{IN} + \mathsf{OUT}^{2+\varepsilon}\mathsf{IN}^{\varepsilon})$, without any assumptions;*

(b) *Work $\mathcal{O}(\mathsf{IN}^{1+\varepsilon} \cdot \mathsf{OUT}^{1+\varepsilon})$ and space $\mathcal{O}((\mathsf{IN} \cdot \mathsf{OUT})^{1+\varepsilon} |D|)$, in the presence of a dictionary.*

In [5, Definition 36] and [8, Definition 3.2], a definition of free-connex, not necessarily acyclic, conjunctive queries is given. Corollary 7.6 can be extended to that class of queries along the lines of [8, Lemma 4.4].

We plan to give a more detailed account in a journal version of this paper.

7.3 Weakly Worst-Case Optimal Work for Natural Joins

This section is concerned with the evaluation of *natural join queries* $q = R_1 \bowtie \ldots \bowtie R_m$ over some schema $\Sigma = \{R_1, \ldots, R_m\}$ with attributes $\mathtt{attr}(q) = \bigcup_{i=1}^m \mathtt{attr}(R_i)$. It was shown in [4] that $|q(D)| \le \prod_{i=1}^m |R_i|^{x_i}$ holds for every database D and that this bound is tight for infinitely many databases D (this is also known as the AGM bound). Here $x_1, \ldots, x_m$ is a fractional edge cover of q defined as a solution of the following linear program.

$$\text{minimize} \sum_{i=1}^m x_i \text{ subject to} \sum_{i: A \in \mathtt{attr}(R_i)} x_i \ge 1 \text{ for all } A \in \mathtt{attr}(q)$$
$$\text{and } x_i \ge 0 \text{ for all } 1 \le i \le m$$

We say that a natural join query q has *weakly worst-case optimal* $\mathcal{O}(1)$-time parallel algorithms, if, for every $\varepsilon > 0$, there is a $\mathcal{O}(1)$-time parallel algorithm that evaluates q with work $(\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})^{1+\varepsilon}$. For comparison, in the sequential setting, algorithms are considered worst-case optimal if they have a time bound $O(\prod_{i=1}^m |R_i|^{x_i} + \mathsf{IN})$ [23]. In this subsection, we show that natural join queries indeed have weakly worst-case optimal $\mathcal{O}(1)$-time parallel algorithms.

▶ **Theorem 7.7.** *For every $\varepsilon > 0$ and natural join query $q = R_1 \bowtie \ldots \bowtie R_m$ with attributes $X = (A_1, \ldots, A_k)$, there is a $\mathcal{O}(1)$-time parallel algorithm that, given arrays $\mathcal{A}_{R_1}, \ldots, \mathcal{A}_{R_m}$ ordered w.r.t. X, computes $q(D)$ and requires $\mathcal{O}\left(\left((\prod_{i=1}^m |R_i|^{x_i}) + \mathsf{IN}\right) \cdot \mathsf{IN}^{\varepsilon}\right)$ work and space on a CRCW-PRAM where $(x_1, \ldots, x_m)$ is a fractional edge cover of q.*

Proof idea. A $\mathcal{O}(1)$-time parallel algorithm can proceed, from a high-level perspective, similarly to the sequential attribute elimination join algorithm, see e.g. [3, Algorithm 10].

In a nutshell, the algorithm computes iteratively, for increasing j from 1 to k relations L_j defined as follows: $L_1 = \bigcap_{1 \le i \le m, A_1 \in \mathtt{attr}(R_i)} \pi_{A_1}(R_i)$ and, for $j > 1$, L_j is the union of all relations $V_t = \{t\} \times \bigcap_{1 \le i \le m, A_j \in \mathtt{attr}(R_i)} \pi_{A_j}(R_i \ltimes \{t\})$ for each $t \in L_{j-1}$. L_k is then the query result $q(D)$. Note that each L_j contains tuples over attributes $X_j = (A_1, \ldots, A_j)$.

To achieve the desired running time in the sequential setting, it is essential that each relation V_t for $t \in L_{j-1}$ is computed in time $\tilde{\mathcal{O}}(\min_{1 \le i \le m} |R_i \ltimes \{t\}|)$, where $\tilde{\mathcal{O}}$ hides a logarithmic factor; for instance with the Leapfrog algorithm, see e.g. [27], [3, Proposition 27.10].

In the parallel setting each relation V_t is computed with work $\mathcal{O}(\min_{1 \le i \le m} |R_i \ltimes \{t\}| \cdot \mathsf{IN}^{\frac{1}{2}\varepsilon})$ – for all tuples $t \in L_{i-1}$ in parallel. Note that the work bound is *not* uniform, i.e. the work bound for a tuple t depends on how many "matching" tuples there are in each of the input relations. This makes assigning processors challenging.

Utilizing that the input relations are ordered w.r.t. X_j, our algorithm groups the tuples in the relations $\pi_{X_j}(R_i)$ w.r.t. X_{j-1} and identifies, for each $t \in L_{j-1}$, the corresponding group in $\pi_{X_j}(R_i)$. These groups are compacted using $\mathtt{Compact}_\delta$ for $\delta = \frac{\varepsilon}{4}$ which allows to approximate the size of $R_i \ltimes \{t\}$ up to a factor of $\mathsf{IN}^{\frac{1}{2}\varepsilon}$ for each i, and, thus, $\min_{1 \le i \le m} |R_i \ltimes \{t\}|$ for each tuple t.

The tuples in L_{j-1} are then partitioned w.r.t. (the approximation of) $\min_{1 \le i \le m} |R_i \ltimes \{t\}|$ into sets $S_{j,\ell}$. Each tuple in a set $S_{j,\ell}$ can then be assigned the same number of processors, determined by the size of the array for the smallest group, similarly as in the Leapfrog algorithm. This is feasible because the number of sets $S_{j,\ell}$ in the partition is bounded by a constant due to the guarantees of $\mathtt{Compact}_\varepsilon$.

The full proof is given in the full version of this paper [19]. ◀

We plan to address the evaluation of natural join queries in the dictionary setting in a journal version of this paper. We expect that almost the same work bound holds, with an additional summand $\mathcal{O}(\max_{1 \le i \le m} |R_i|^2)$, accounting for the grouping of each $\pi_{X_j}(R_i)$ with respect to $\pi_{X_{j-1}}(R_i)$.

8 Conclusion

This paper is meant as a first study on work-efficient $\mathcal{O}(1)$-time parallel algorithms for query evaluation and many questions remain open. The results are very encouraging as they show that quite work-efficient $\mathcal{O}(1)$-time parallel algorithms for query evaluation are possible. In fact, the results give a hint at what could be a good notion of *work-efficiency* in the context of constant-time parallel query evaluation. Our impression is that work-optimality is very hard to achieve in constant time and that query evaluation should be considered as work-efficient for a query language, if there are constant-time parallel algorithms with $\mathcal{O}(T^{1+\varepsilon})$ work, for every $\varepsilon > 0$, where T is the best sequential time of an evaluation algorithm. Of course, it would be nice if this impression could be substantiated by lower bound results, but that seems to be quite challenging.

We have not given results for all combinations of query languages and settings, e.g., Subsection 7.1 and Subsection 7.2 do not yet cover the ordered setting and Subsection 7.3 not the dictionary setting.

As mentioned in Section 3, when finding the results of this paper we were unaware of the fact that [13] provides algorithms for *ordered* compaction with constant time and work $\mathcal{O}(n^{1+\varepsilon})$. Naturally, these algorithms can be useful for the ordered setting and we expect them to yield a $\mathcal{O}(n^{1+\varepsilon})$ work bound for the semi-join algebra (Subsection 7.1). We do not expect them to improve the bounds for natural joins (Subsection 7.3) or for general acyclic queries (Subsection 7.2). We plan to fully explore the consequences in a journal version of this paper, but we decided against incorporating them into the final version of this paper, due to the lack of peer-review. In that journal version we will also address some of the reviewer's suggestions that could not be incorporated yet.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Miklós Ajtai. Σ_1^1 formulae on finite structures. *Ann. of Pure and Applied Logic*, 24:1–48, 1983. `doi:10.1016/0168-0072(83)90038-6`.

3 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory.* Open source at `https://github.com/pdm-book/community`, 2021. Preliminary Version, August 19, 2022.

4 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. `doi:10.1137/110859440`.

5 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. `doi:10.1007/978-3-540-74915-8_18`.

6 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990. `doi:10.1016/0022-0000(90)90022-D`.

7 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017. `doi:10.1145/3125644`.

8 Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020. `doi:10.1145/3385634.3385636`.

9 Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre.* Theses, Université de Caen, April 2013. URL: `https://hal.archives-ouvertes.fr/tel-01081392`.

10 Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 271–282. ACM, 2001. `doi:10.1145/375663.375692`.

11 E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 33–64. Prentice-Hall, 1972.

12 Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984. `doi:10.1007/BF01744431`.

13 Tal Goldberg and Uri Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Third Israel Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings*, pages 220–228. IEEE Computer Society, 1995. `doi:10.1109/ISTCS.1995.377028`.

14 Torben Hagerup. On a compaction theorem of Ragde. *Inf. Process. Lett.*, 43(6):335–340, 1992. `doi:10.1016/0020-0190(92)90121-B`.

15 Xiao Hu and Ke Yi. Massively parallel join algorithms. *SIGMOD Rec.*, 49(3):6–17, 2020. `doi:10.1145/3444831.3444833`.

16 Neil Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18(3):625–638, 1989. `doi:10.1137/0218043`.

17 Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999. `doi:10.1007/978-1-4612-0539-5`.

18 Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

19 Jens Keppeler, Thomas Schwentick, and Christopher Spinrath. Work-efficient query evaluation with PRAMs, 2023. `doi:10.48550/ARXIV.2301.08178`.

20 Paraschos Koutris, Semih Salihoglu, and Dan Suciu. Algorithmic aspects of parallel data processing. *Found. Trends Databases*, 8(4):239–370, 2018. `doi:10.1561/1900000055`.

21 Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005. `doi:10.1007/s10849-005-5789-8`.

22 Philip D. MacKenzie. Load balancing requires omega($\log^* n$) expected time. In Greg N. Frederickson, editor, *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida, USA*, pages 94–99. ACM/SIAM, 1992. URL: `http://dl.acm.org/citation.cfm?id=139404.139425`.

23 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. `doi:10.1145/3180143`.

24 Benjamin Rossman. On the constant-depth complexity of k-clique. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 721–730, 2008. `doi:10.1145/1374376.1374480`.

25 Jonas Schmidt, Thomas Schwentick, Till Tantau, Nils Vortmeier, and Thomas Zeume. Work-sensitive dynamic complexity of formal languages. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021*, volume 12650 of *Lecture Notes in Computer Science*, pages 490–509. Springer, 2021. `doi:10.1007/978-3-030-71995-1_25`.

26 Peter van Emde Boas. Machine models and simulation. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1990.

27 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014. `doi:10.5441/002/icdt.2014.13`.

28 Yilei Wang and Ke Yi. Query evaluation by circuits. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 67–78. ACM, 2022. doi:10.1145/3517804.3524142.

29 Mihalis 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.