

On Optimizing Relational Self-Joins

Yu Cao^{†‡}, Yongluan Zhou[§], Chee-Yong Chan[†], Kian-Lee Tan[†]

[†]*School of Computing, National University of Singapore, Singapore*
{caoyu, chancy, tankl}@comp.nus.edu.sg

[‡]*EMC Labs, China*
yu.cao@emc.com

[§]*Department of Mathematics and Computer Science, University of Southern Denmark, Denmark*
zhou@imada.sdu.dk

ABSTRACT

Self-join, which joins a relation with itself, is a prevalent operation in relational database systems. Despite its wide applicability, there has been little attention devoted to improving its performance. In this paper, we present **SCALE** (Sort for Clustered Access with Lazy Evaluation), an efficient self-join algorithm, which takes advantage of the fact that both inputs of a self-join operation are instances of the same relation. **SCALE** first sorts the relation on one join attribute, say $R.A$. In this way, for every value of the other join attribute, say $R.B$, its matching $R.A$ tuples are essentially clustered. As **SCALE** scans the sorted relation, each tuple is joined with its matching tuples co-existing in memory. For tuples where full-range clustered accesses to their matching tuples are not possible, they are buffered and the unfinished part of join processing deferred. Such lazy evaluation minimizes the need for “random” access to the matching tuples. **SCALE** further optimizes the memory allocation for clustered access and lazy evaluation to keep the processing cost minimal. Our analytical study shows that **SCALE** degenerates gracefully to a Sort-Merge Join in the worst case. We have also implemented **SCALE** in PostgreSQL, and results of our extensive experimental study show that it outperforms both Sort-Merge Join and Hybrid Hash Join by a wide margin in (almost) all cases.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—query processing, relational databases

General Terms

Algorithms, Design, Performance

*Research partially done while visiting University of Southern Denmark.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT 2012, March 26–30, 2012, Berlin, Germany.

Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...\$10.00

Keywords

self-join, join operation, join algorithms, query optimization

1. INTRODUCTION

Self-join is a join operation that relates data within a relation by joining the relation with itself. We consider self-joins with the join predicates involving two distinct attributes (i.e., $R_1.A \text{ op } R_2.B$, where R_1 and R_2 are instances of relation R). We observe that this type of self joins occur frequently in many recently emerging database applications, such as location-based service (LBS), RFID data management, sensor networks, network management etc. Below are two examples:

Example 1: Consider a database of moving objects with a user-defined view, *TRAJECTORY*, storing the line segments of the trajectories of many moving objects. The schema of *TRAJECTORY* is (*objID*, *location1*, *location2*, *time1*, *time2*). Each record describes the movement of an object from *location1* at *time1* to *location2* at *time2*. An example *LBS* query is: for each location L , return all the pairs of objects $O1$ and $O2$, such that $O1$ arrived at L shortly, say within t time units, before $O2$ moved out of L . This query involves a self equi-join condition:

```
SELECT *
FROM TRAJECTORY O1, TRAJECTORY O2
WHERE O1.location2 = O2.location1
      AND O1.time2 > O2.time1 - t
      AND O1.time2 < O2.time1
```

Example 2: Consider a table storing the readings from a network of temperature sensors with schema: (*sensorId*, *temperature*, *timestamp*). A view *STATS* is created on top of the table with schema: (*sensorId*, *hourId*, *avgTemp*, *maxTemp*, *minTemp*). Each row in *STATS* provides the average, maximum and minimum temperatures reported by a sensor over a particular hour of the day. A user may be interested in finding out the pairs of sensors, $S1$ and $S2$, such that the average temperature reported by $S1$ is equal to or lower by at most 2 degrees than the minimum temperature reported by $S2$, within a certain time proximity, say one hour. This query involves two self band-join [6, 13] conditions¹:

```
SELECT *
FROM STATS S1, STATS S2
WHERE S1.avgTemp <= S2.minTemp
```

¹A general band-join condition has the form $R.A - c_1 \leq S.B \leq R.A + c_2$, where c_1 and c_2 are constants but can not both be zero.

```

AND S1.avgTemp >= S2.minTemp - 2
AND S1.hourId <= S2.hourId + 1
AND S1.hourId >= S2.hourId - 1
AND S1.sensorId <> S2.sensorId

```

Moreover, self-join is also common in RDF data management where self-join is used to relate the subjects and objects of a triple table, and in the publication of relational data as XML where XML queries (e.g. XQuery) over XML views are translated into self-joins of base relational tables.

Despite the importance and prevalence of self-joins, there however have been surprisingly few research efforts on optimizing them.

On the one hand, existing solutions either employ join indexes [12] or handle the special case where the join attributes are on the same attribute (e.g., $R_1.A = R_2.A$) [4, 8]. As one can see from the examples, many emerging queries involve self joins on two distinct attributes. While index-based techniques could be applied to the problem, it is possible that indexes do not exist, especially when the queries are ad-hoc and/or the join attributes are derived and computed from user defined functions as shown in Example 2. Even when indexes exist, they may not be useful. For example, if the join selectivity is high (i.e. a lot of join results), then indexes, especially the non-clustered ones, are not beneficial.

On the other hand, conventional join algorithms, such as Sort-Merge Join (SMJ) and Hybrid Hash Join (HHJ), treat the two instances of the same relation as distinct relations. As such, they miss the opportunities to enhance the processing performance, particularly in keeping the I/O cost low.

To improve performance, we need a scheme that can take advantage of the fact that the two inputs of a self-join operator are instances of the same relation. Towards this end, we propose a novel and efficient self equi-join algorithm called **SCALE** (Sort for Clustered Access with Lazy Evaluation), which is also easily extendible to handle self band-joins [6, 13]. **SCALE** first sorts the relation, say R , on one of the join attributes, say A , to produce a sorted sequence, denoted as $S_A(R)$ (when R has already been ordered by some join attribute, this sorting step can be avoided). Now, given a tuple t with $t.B = x$, where B is the other join attribute, all the matching tuples whose $R.A$ value is x are clustered within $S_A(R)$. As such, by scanning $S_A(R)$, we have two possible cases. First, for each tuple t with $t.B = x$, we have a clustered access of all matching tuples if t co-exists with them in memory. In this case, we can generate and produce the join results at no extra cost since this is within a single scan of $S_A(R)$. Second, for a tuple t with $t.B = x$ for which such clustered access is not possible (e.g., matching $R.A$ tuples may not co-exist with t in memory), it is buffered and possibly spilled to disk to defer the join processing to a later time. Such lazy evaluation minimizes the need for “random” accesses to the matching tuples.

To support clustered access and lazy evaluation, the memory space has to be effectively allocated between these two tasks. To optimize performance, **SCALE** adopts a cost-based approach to manage the memory allocation for clustered access and lazy evaluation. **SCALE** is also able to handle the situation where the two joining instances of the same relation are associated with different tuple selection and projection predicates. Moreover, we can improve **SCALE** with sideways information passing techniques to further reduce the cost when many tuples have no join matchings.

Our analytical study shows that **SCALE** degenerates gracefully to Sort-Merge Join (SMJ) in the worst case. We have

also implemented **SCALE** in PostgreSQL [1], and the results of our extensive experimental study confirms our analytical results. Moreover, it shows that **SCALE** outperforms both Sort-Merge Join and Hybrid Hash Join by 20% - 40% in (almost) all cases.

The rest of the paper is organized as follows. Section 2 surveys the related work of this paper. Section 3 discusses the technical details of the **SCALE** algorithm. We then present a thorough analytical study of **SCALE** in Section 4. An extensive experimental study is presented in Section 5. In Section 6, extensions to **SCALE** supporting self band-join and side-ways information passing are proposed. We conclude the paper in Section 7.

2. RELATED WORK

The join operation is one of the most time-consuming and data-intensive operations. Therefore, it is critical to implement joins in the most efficient way possible. The join operation has been studied and discussed extensively in the literature (see [9] for a comprehensive survey). Common ad-hoc join techniques can be classified into three broad categories: nested-loop join, sort-merge join [3] and hash-based join [5, 7, 14]. Recently, Goetz proposed *g-join* [10], a generalized join algorithm, intending to replace the above join techniques. The major advantage of *g-join* over sort-merge join and hash-based join lies in its robustness rather than its potential performance gain. This implies that *g-join*’s costs should be comparable with those of sort-merge join and hybrid hash join. Therefore, in our experiments we did not directly compare with *g-join*. Besides, the *Diag-Join* algorithm [11] optimizes a special case of 1:N join where the two joined tables have the time-of-creation clustering property, which usually does not hold for the single table involved by a self-join. Thus, *Diag-Join* cannot be directly utilized for optimizing self-joins.

Nevertheless, to the best of our knowledge, only a few works specifically focus on self-join processing. In [12], Lei and Ross proposed the *Stripe* algorithm for performing a join with a *join index* [16], which maintains pairs of identifiers of tuples that would match in case of a join between two relations. *Stripe* join was designed for general join processing but is particularly efficient for self-joins. However, the applicability of *Stripe* join is highly dependent on the availability of the suitable join index, which must have been materialized and maintained by the database systems before the join execution. In contrast, our algorithm is more useful in application contexts identified in Section 1, e.g. the join attribute is a derived one that is not indexed. The problem of self-join size estimation has been tackled in both centralized [2] and large-scale distributed [15] database systems.

In this paper, we mainly discuss the situation that the self-join predicate involves two distinct attributes. In the special case where the join predicate is an equi-join over the same attribute (e.g., $R_1.A = R_2.A$), the join may be evaluated by partitioning the base relation according to the attributes involved in the equality join predicates and then performing a simplified join operation separately on each partition. The identification and execution of such a special case of self-join were addressed previously in [4, 8]. Our algorithm is actually applicable to such special cases and essentially behaves like the partition-based evaluation strategies in [4, 8]. Therefore, the performances are expected to be comparable for such special cases.

3. THE SCALE ALGORITHM

In this section, we consider a self equi-join $R_1.A = R_2.B$. Both R_1 and R_2 are instances of the same relation R , where A and B are two (single or composite) attributes.

In the self-join, each tuple t in R is associated with two sets of matching tuples referred to as its *left-matching* and *right-matching* tuples. A tuple t' is a left-matching (resp. right-matching) tuple of t if $t.A = t'.B$ (resp. $t'.A = t.B$). We define the *left-join* (resp. *right-join*) of t to be the result of joining t with its left-matching (resp. right-matching) tuples. Thus, the self-join of R can be computed as the union of the left-join of each tuple in R or symmetrically as the union of the right-join of each tuple in R .

3.1 Overview

Without loss of generality, we present our self-join evaluation algorithm **SCALE** in terms of the union of the right-join of each tuple. The choice between a left-join or right-join evaluation of R can be decided in a cost-based manner using the cost model in Section 4.1.

To evaluate the self-join of R in terms of right-joins, **SCALE** first sorts R in ascending order of A to produce the sorted table $S_A(R)$ (sorting is unnecessary if R has already been ordered by A due to the existence of clustered indices). As such, for each tuple in R , all its right-matching tuples are clustered together in $S_A(R)$. **SCALE** then processes $S_A(R)$ in at most two passes as follows. In the first pass, **SCALE** maintains three main-memory buffers, namely, *main*, *hold*, and *defer* buffers. **SCALE** sequentially scans the tuples in $S_A(R)$ into the main buffer and computes the right-joins between the newly scanned tuple and the existing tuples in the main and hold buffers. The main buffer is managed using a replacement policy to evict tuples when the buffer becomes full. Due to tuple evictions, the right-join of a tuple t could be partially processed when t is evicted out of the main buffer. Thus, at the same time that a tuple t is being evicted from the main buffer, we can classify t into one of three possible states (*complete*, *prefix-complete*, or *incomplete*) as follows. If the right-join of t has been completed, t is classified as complete; otherwise, if the right-join between t and all its right-matching tuples that precede t in $S_A(R)$ has been completed, t is classified as prefix-complete; otherwise, t is classified as incomplete.

Before evicting a tuple t from the main buffer, **SCALE** first determines the state of the right-join of t . If t is complete, then t is simply evicted from the main buffer; otherwise, t needs to be buffered elsewhere (either in the hold or in the defer buffer) for subsequent processing of its remaining right-join. Specifically, if t is incomplete, then t is transferred to the defer buffer; otherwise, t must be prefix-complete and it is transferred to the hold buffer. The tuples in the hold buffer will “wait” for their unread right-matching tuples in $S_A(R)$ to be scanned into the main buffer to complete their right-join processing during the first pass. The tuples in the defer buffer will complete their right-join computation in the second pass.

At the end of the first pass, if the defer buffer is empty, this means the self-join of R has been completely evaluated and **SCALE** will therefore terminate; otherwise, **SCALE** will proceed with the second pass to process the tuples in the defer buffer.

In the second pass, **SCALE** computes the remaining right-join of each tuple in the defer buffer by performing a merge

join of the tuples in $S_A(R)$ and the defer buffer. The tuples in $S_A(R)$ (which are already sorted on A) will be scanned sequentially to merge with the tuples in the defer buffer which will be retrieved in ascending order of B .

To facilitate the right-join processing, both the hold and defer buffers are organized as min-heaps ordered on the B values. When the hold/defer buffer overflows, **SCALE** flushes a sorted run from the appropriate heap to the disk. Thus, **SCALE** will subsequently need to load back the disk-based sorted-runs during processing the right-joins, and an additional buffer, referred to as the *run* buffer, is used for this purpose.

It is important that a join computation that has been performed in the first pass is not computed again during the second pass. **SCALE** ensures this by simply recording for each tuple t spilled to the defer buffer, the ranks of the first and last tuples in $S_A(R)$ (denoted by $first(t)$ and $last(t)$ respectively) that right-join with t during the first pass. During the second pass, for each tuple t in the defer buffer, **SCALE** computes the right-join of t with a right-matching tuple t' in $S_A(R)$ if and only if the rank of t' either precedes $first(t)$ or succeeds $last(t)$. For correctness, the tuple eviction policy of the main buffer ensures that if two tuples t and t' in the main buffer have the same A values and t is evicted before t' , then t must precede t' in $S_A(R)$. Thus, **SCALE** correctly computes the self-join of R without missing out any join result and without computing the same join more than once.

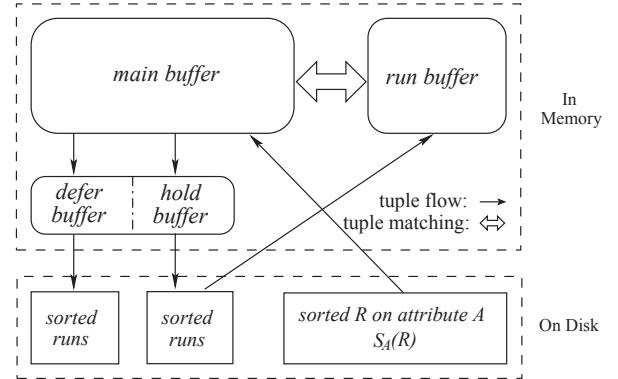


Figure 1: SCALE execution during the first pass of processing $S_A(R)$

Fig. 1 shows the execution of **SCALE** during the first pass of processing $S_A(R)$. The functions of each component, the tuple flow and the tuple matching procedure will be elaborated in the following subsection.

3.2 Algorithm Details

For a tuple t , we denote the set of its right-matching tuples in $S_A(R)$ by $RM(t)$. During the first pass of processing $S_A(R)$, tuples in $RM(t)$ can be divided into three subsets:

- $RM_1(t)$: the tuples in $RM(t)$ that have left the main buffer when t is read into the main buffer.
- $RM_2(t)$: the tuples in $RM(t)$ that will meet and join with t in the main buffer.
- $RM_3(t)$: the tuples in $RM(t)$ that will only be read into the main buffer after t has left the main buffer.

For a tuple t currently in the main buffer, if it is evicted, there will be four possible cases according to the distribution

case #	$RM_1(t)$	$RM_3(t)$	right-join state of t
1	empty	empty	complete
2	non-empty	empty	incomplete
3	non-empty	non-empty	incomplete
4	empty	non-empty	prefix-complete

Table 1: The possible distribution of $RM(t)$ tuples within $RM_1(t)$ and $RM_3(t)$, along with the corresponding right-join state of t

of $RM(t)$ tuples within $RM_1(t)$ and $RM_3(t)$, as illustrated in Table 1.

3.2.1 Tuple Eviction Policy of the Main Buffer

Whenever the main buffer becomes full, each tuple inside is classified into one of the above four cases and is assigned with an eviction priority. Tuples with higher priorities will get evicted first. The destination of an evicted tuple is dependent on its right-join state, as discussed in Section 3.1. The ultimate goal of the tuple eviction policy is to improve tuples' clustered access to their right-matching tuples and thus maximize the total number of tuples reaching a complete right-join state. Besides, a secondary goal is to produce early join results at high rates. We describe our tuple eviction policy by comparing the eviction priorities of two arbitrary tuples t and t' .

First of all, when $t.A = t'.A$, if t precedes t' in $\mathcal{S}_A(R)$, then t has a higher eviction priority than t' . This rule ensures the correctness of **SCALE** as discussed at the end of Section 3.1. When $t.A \neq t'.A$, the following heuristic rules are applied:

- When t is in cases 1 or 2 and t' is in cases 3 or 4, t has a higher eviction priority than t' , since by staying in the main buffer t' could continue joining with more $RM_3(t')$ tuples being or to be read from $\mathcal{S}_A(R)$ and thus is likely to be classified into cases 1 or 2 when it is evicted in the future.
- When both t and t' are in one of cases 1 and 2, they have equal eviction priorities.
- When t is in case 3 and t' is in case 4, t has a higher eviction priority than t' , as by staying in the main buffer t' still has the chance to reach the complete state later.
- When both t and t' are in case 3, they have equal eviction priorities.
- When both t and t' are in case 4, t has a higher eviction priority than t' if $t.B > t'.B$, since as such $RM_3(t')$ tuples will be read earlier than $RM_3(t)$ tuples and thus by staying in the main buffer t' has a bigger chance to reach the complete status later.

It is obvious that the above rules will cover all scenarios between t and t' , and thus can produce a global ranking of the eviction priorities of all tuples in the main buffer.

3.2.2 Processing Tuples in the Hold Buffer

For each tuple t transferred from the main buffer to the hold buffer, $RM_1(t)$ is empty and $RM_3(t)$ is non-empty. During the first pass of processing $\mathcal{S}_A(R)$, t will wait in the hold buffer until the $RM_3(t)$ tuples are sequentially scanned into the main buffer, and then t will be moved into the run buffer so as to complete its remaining right-join processing with the $RM_3(t)$ tuples².

²In the hold buffer, multiple tuples could share the same attribute B value. If their total size is larger than the size of the run buffer, then the join processing for them degrades to a nested loop join, and each tuple has to be moved into the run buffer more than once.

Since the $\mathcal{S}_A(R)$ tuples are sorted on A , tuples in the hold buffer need to be retrieved and processed in the order of their B values. To achieve this, the hold buffer is organized as a min-heap ordered on the B values and is used to generate disk-based sorted tuple runs when buffer overflows. Once some tuples in the hold buffer need to join with their remaining right-matching tuples newly read into the main buffer, they are retrieved into the run buffer by progressively reading and (recursively) merging the sorted runs, while the min-heap may be simultaneously dumping and appending tuples to some existing or new runs on the disk. Fig. 2 shows the procedure of flushing hold buffer tuples from the min-heap to the sorted runs and then reading them back to the run buffer.

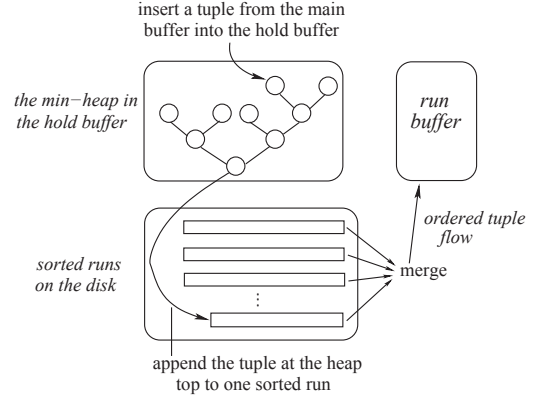


Figure 2: Insert tuples to the hold buffer as well as read them into the run buffer

3.2.3 Processing Tuples in the Defer Buffer

For each tuple t transferred from the main buffer to the defer buffer, at least $RM_1(t)$ is non-empty. Since $RM_1(t)$ tuples precede t in $\mathcal{S}_A(R)$, t is not able to join with $RM_1(t)$ tuples during the first pass of processing $\mathcal{S}_A(R)$. As a result, t will only complete its right-join with tuples in $RM_1(t)$ and $RM_3(t)$ after the first pass of processing $\mathcal{S}_A(R)$ ends, via a merge join of the tuples in $\mathcal{S}_A(R)$ and the tuples in the defer buffer. For the purpose of merge join, $\mathcal{S}_A(R)$ tuples will be sequentially scanned once again into the main buffer, and defer buffer tuples will be read into the run buffer in the order of their B values. Similarly to the hold buffer, we organize the defer buffer as a min-heap ordered on the B values and generate disk-based sorted tuple runs when buffer overflows. However, although generated during the first pass of processing $\mathcal{S}_A(R)$, these sorted runs are (recursively) merged in the run buffer during the second pass.

Note that actually tuples in the hold buffer can be processed together with tuples in the defer buffer via the merge join after the first pass of processing $\mathcal{S}_A(R)$. However, processing tuples in the hold buffer during the first pass can incur less I/O cost as analyzed in Section 4.1, and can generate join results earlier.

3.2.4 Memory Allocation for Buffers

In the second pass of processing $\mathcal{S}_A(R)$, **SCALE** only maintains the main buffer and the run buffer to conduct the merge join. Both buffers dynamically share all available memory space. As such, the run buffer may be able to grab enough memory to conduct a single-step merge of sorted runs of defer buffer tuples.

In the first pass, both the main buffer and the run buffer will have predetermined sizes, while the hold buffer and the defer buffer will dynamically share the remaining memory space so as to maximize the average lengths of (and minimize the total number of) generated sorted runs. We thereby develop a cost-based heuristic to optimize the memory allocation for buffers in this pass.

Suppose the size of R , in terms of the number of pages, is N and the total available join memory is M pages from R . The sizes of tuples in the hold buffer and in the defer buffer are N_1 and N_2 respectively. The memory allocation scenario would affect the I/O costs incurred by tuples in the hold and defer buffers most significantly. In Section 4.1, we theoretically analyze that N_1 would be small and thus the memory allocation of the run buffer is not critical to the performance. Thus, we will experientially predetermine a small size for the run buffer. In so doing, the rest of our work is simplified to finding a good memory distribution between the main buffer (with a size M_s), and the hold and defer buffers (with a total size M_f). We use M' to denote the amount of memory available for M_s and M_f , i.e. $M_s + M_f = M'$.

From the analysis in Section 4.1, we can see that the values of N_1 and N_2 depend on M_s . Basically, a higher M_s can produce smaller N_1 and N_2 . On the other hand, M_f affects the number of run merging steps for the defer buffer. In general, a smaller M_f may increase the number of run merging steps for the defer buffer.

The cost model derived in Section 4.1 estimates N_1 and N_2 based on M_s and then calculates the total I/O cost of SCALE as

$$2N(\lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 2) + 2N_1 + 2N_2(\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) \quad (1)$$

The objective of memory allocation is to minimize the total cost shown in Eqn. (1). To simplify the problem, we assume a linear relationship between M_s and the values of N_1 and N_2 . With this assumption, we could estimate N_1 and N_2 with different M_s values as follows. First, we use the cost model to estimate N_1 (or N_2) under the two situations of $M_s = M'$ and $M_s = 0$. The values are denoted as N_{11} (or N_{21}) and N_{12} (or N_{22}) respectively. Then given any $M_s \leq M'$, we can estimate N_1 and N_2 as

$$N_1 = (M' - M_s)(N_{12} - N_{11})/M' + N_{11} \quad (2)$$

$$N_2 = (M' - M_s)(N_{22} - N_{21})/M' + N_{21} \quad (3)$$

The memory allocation algorithm is based on the following observation of Eqn. (1): the dominant impact that the increase of M_f is probably able to make is reducing the value of the term $f(M_f) = \lceil \log_M \lceil N_2/(2M_f) \rceil \rceil$. The algorithm starts by setting the M_f to a minimum value, say 1. Then it iteratively attempts to set M_f to a higher number such that the value of $f(M_f)$ decreases by 1 in each iteration. If this results in a lower total cost of Eqn. (1), then the attempts will be continued. Otherwise the loop will stop. The loop will also stop if $f(M_f)$ already reaches 1. As in practice the value of $f(M_f)$ tend to be very small due to the logarithmic effect, the loop will stop after several iterations.

3.3 Integration with Tuple Selection and Projection Pushdown

For a self-join between two instances R_1 and R_2 of relation R , it is possible that each instance additionally involves distinct tuple filtering and projection conditions. In the conventional query processing, tuple selection and projection operations are usually *pushed down* onto the lowest

feasible levels within the query execution tree. As a result, the physical join implementation of the self-join has to deal with two input sets of tuples, each of which is (horizontally and/or vertically) a subset of R . Suppose the tuple selection and projection attached to R_i ($i \in \{1, 2\}$) are σ_i and Π_i respectively, and suppose the tuple projection attached to the self-join operation is Π , the algebra expression of the self-join is hereby $\Pi((\Pi_1(\sigma_1(R_1))) \bowtie (\Pi_2(\sigma_2(R_2))))^3$.

According to σ_1 and σ_2 , the tuples of R that are relevant to the self-join can be classified into three categories: satisfying σ_1 only, satisfying σ_2 only and satisfying both σ_1 and σ_2 , which are denoted with C_{σ_1} , C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ respectively. The filtered R_1 consists of C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples, while the filtered R_2 consists of C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ tuples. Clearly, the (un-projected) self-join result can be represented by $(C_{\sigma_1} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1} \bowtie C_{\sigma_1 \cap \sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_1 \cap \sigma_2})$.

In order to evaluate a self-join integrated with tuple selection and projection pushdown, a straightforward extension to SCALE works as follows. As the first step, we sort R on A to obtain a $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, which contains a sorted sequence of all above three categories of tuples. In addition, in $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, the C_{σ_1} , C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ tuples are projected by Π_1 , Π_2 and $\Pi_1 \cup \Pi_2$ respectively. We then run the rest of the algorithm as usual, with the mere modifications that C_{σ_1} (resp. C_{σ_2}) tuples are distinguished to act only as the left-hand (resp. right-hand) side of the self-join, and that C_{σ_1} tuples behave as if they had no right-matching tuples when they are considered for tuple eviction in the main buffer. However, there are several potential problems with such a straightforward extension. First of all, intuitively it incurs wasteful I/O and CPU costs to sort C_{σ_2} tuples on A , as C_{σ_2} tuples are not a part of right-matching tuples. Moreover, during the first pass of processing $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, C_{σ_2} tuples will keep C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples farther away from their right-matching tuples. As such, more C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples would be forced to enter the hold buffer and the defer buffer and incur additional I/O overhead.

There is another more efficient approach. The rough idea is to split those three categories of tuples in R into two parts: one part R^+ contains all C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples, and the other part R^- contains the rest C_{σ_2} tuples. Similarly, the C_{σ_1} tuples in R^+ are projected by Π_1 , the $C_{\sigma_1 \cap \sigma_2}$ tuples in R^+ are projected by $\Pi_1 \cup \Pi_2$ and the C_{σ_2} tuples in R^- are projected by Π_2 . Given the self-join condition $R_1.A = R_2.B$, we need to sort R^+ on A into $\mathcal{S}_A(R^+)$ and sort R^- on B into $\mathcal{S}_B(R^-)$. We then sequentially read both $\mathcal{S}_A(R^+)$ and $\mathcal{S}_B(R^-)$ into memory to merge join them, which generates the projected result tuples of $(C_{\sigma_1} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_2})$. In the meantime, we also apply the self-join techniques of SCALE to $\mathcal{S}_A(R^+)$ to produce the projected result tuples of $(C_{\sigma_1} \bowtie C_{\sigma_1 \cap \sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_1 \cap \sigma_2})$.

4. ANALYTICAL STUDY

In this section, we first analyze the cost of the SCALE algorithm and then analytically prove that the performance of SCALE is at least as good as Sort-Merge join.

4.1 Cost Model

Our SCALE algorithm on join condition $R.A = R.B$ is sym-

³Note that the query optimizer will ensure Π_1 contains A and Π_2 contains B , even if Π does not contain A and/or B .

metric: it could choose to sort on either A or B during the first-step external sorting. However, this choice may affect the final total join cost and thus should be decided in a cost-based way. Moreover, the query optimizer also needs a cost model to estimate the self-join subplan costs when doing join enumeration and pruning.

Notation	Definition
N	the size of R in terms of pages
M	the total number of buffer pages available for join
M_s	the total number of pages occupied by main buffer
M	the number of R tuples that can be held by main buffer
M_f	the number of pages occupied by hold and defer buffers
N_1	the total size of tuples transferred to hold buffer
N_2	the total size of tuples transferred to defer buffer
$t(x, y)$	a tuple such that $t.A = x$ and $t.B = y$
$N(x, y)$	the number of tuples in R that have values x and y on the attributes A and B respectively
$N_A(x)$	the number of tuples in R that have value x on A
$N_B(y)$	the number of tuples in R that have value y on B
P_t	the position of tuple t in $S_A(R)$
$P_{RM(t)}^{1st}$	the position of the first tuple of $RM(t)$ in $S_A(R)$
$P_{RM(t)}^{last}$	the position of the last tuple of $RM(t)$ in $S_A(R)$

Table 2: Notations used in the analytical study of SCALE

Without loss of generality, in the following discussions, we assume that **SCALE** sorts R on attribute A during the first-step external sorting. Table 2 summarizes the notations used throughout the analytical study of **SCALE**. Generally, the I/O cost of **SCALE** consists of the following components:

- The cost of externally sorting R into $S_A(R)$.
- The cost of sequentially scanning $S_A(R)$ during the first pass of processing $S_A(R)$.
- The cost of inserting tuples into the hold buffer and the defer buffer (i.e. generating sorted runs) during the first pass of processing $S_A(R)$.
- The cost of reading and merging sorted runs of hold buffer tuples during the first pass of processing $S_A(R)$.
- The cost of the merge join of defer buffer tuples and $S_A(R)$ tuples during the second pass of processing $S_A(R)$.

In the ideal situation, the I/O cost of **SCALE** consists of only (a)–(b). Suppose the size of R , in terms of the number of pages, is N and the total available join memory is M pages from R . Then cost (a)–(b) can be calculated as $2N(\lceil \log_M \lceil N/2M \rceil \rceil + 1) + N$.

To calculate (c)–(e), we first assume that we can estimate the total sizes of tuples in the hold buffer and the defer buffer, denoted as N_1 and N_2 respectively. We will show later how to estimate them.

Cost incurred by the hold buffer. The tuples spilled into the hold buffer are stored in a number of disk-based run files sorted on attribute B , which have to be merged on-the-fly using the run buffer during the first pass. Theoretically, we can calculate the number of merging steps based on the size of the run buffer and the number of sorted runs in the hold buffer. However this will result in an overestimation of the actual merging steps due to the following reasons:

- During the first pass of processing $S_A(R)$, as tuples are read into the main buffer in the order of A , roughly the B values of the case 4 tuples being spilled to the hold buffer should be in a nearly sorted order. This is because for such a case 4 tuple with $B = b$, it then must have some matching tuples with $A = b$ that have not been read into the main buffer. Consequently, a few (but large) run files will be generated.
- The runs in the hold buffer are merged progressively while the tuples are being spilled. In other words, at any moment during the process, we are only merging up to the tuples that have been spilled so far.
- As one will see in the later analysis, compared with the FIFO tuple eviction policy for the main buffer, the number of tuples spilled to the hold buffer is significantly reduced by our prioritized tuple eviction policy described in Section 3.2.1.

Therefore, in our cost model, we assume that the tuples in the hold buffer are written to disk and read into memory only once. Our experiment results validate this assumption. Furthermore, this assumption simplifies the cost estimation and saves the optimization cost.

Cost incurred by the defer buffer. The tuples in the defer buffer are also stored as disk-based run files sorted on B and need to be merged during the second pass. The size of each run depends on the size of the total memory, denoted as M_f , that is dynamically shared by the hold buffer and the defer buffer. As mentioned above, the tuples are spilled to the hold buffer in a nearly sorted order and thus require only a small hold buffer. Therefore, we can assume that almost the whole M_f is allocated to the defer buffer. Given the size of M_f , the expected number of runs will be $\lceil N_2/2M_f \rceil$. Furthermore, we can use nearly all the available join memory to merge defer buffer tuples during the second pass. Then the number of steps of run merging defer buffer tuples is $\lceil \log_M \lceil N_2/2M_f \rceil \rceil$. Finally, we also need to count the cost of writing the sorted runs in the defer buffer to the disk before the merging.

In summary, the cost components (c)–(e) can be calculated as follows:

$$2N_1 + N_2(2\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) + N_2 + N \quad (4)$$

and hence the total cost of our algorithm is

$$2N(\lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 2) + 2N_1 + 2N_2(\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) \quad (5)$$

4.1.1 Cost with FIFO Tuple Eviction Policy

Below we discuss how to estimate the values of N_1 and N_2 , i.e. the sizes of tuples spilled to the hold buffer and the defer buffer respectively. Due to the dynamic behavior of our algorithm, the exact estimation is quite complicated and costly to perform. Hence, we perform a simplified analysis by assuming that the main buffer applies the FIFO tuple eviction policy. Moreover, we assume a tuple $t(x, y)$ is randomly located within the segment of tuples with $A = x$ in $S_A(R)$. As such, there are three scenarios under which tuples have to be spilled into the hold buffer and the defer buffer.

(I) If a tuple $t(x, x)$ belongs to case 2, 3 or 4, then it has to be spilled to either the hold buffer or the defer buffer. As t in this case is located inside its own $RM(t)$ in $S_A(R)$, we have $\mathbb{P}_{RM(t)}^{1st} \leq \mathbb{P}_t \leq \mathbb{P}_{RM(t)}^{last}$. Hence, the probability that $t(x, x)$ falls into case 2 or 3 and thus is spilled to the defer buffer can be calculated as follows:

$$P(t(x, y) \text{ is spilled to the defer buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ 1 - \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (6)$$

Furthermore, the probability that $t(x, x)$ belongs to case 4 and thus is spilled to the hold buffer is:

$$P(t(x, y) \text{ is spilled to the hold buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \mathbb{N}_A(x) \geq 2\mathbb{M} \\ 1 - \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (7)$$

(II) With $x > y$, a tuple $t(x, y)$ cannot be in cases 3 and 4. Thus, $t(x, y)$ will not be spilled to the hold buffer, and $P(t(x, y) \text{ is spilled to the hold buffer} \mid x > y) = 0$. If $t(x, y)$ with $x > y$ is in case 2, then it has to be spilled to the defer buffer. Note that, in this case, $\mathbb{P}_t \leq \mathbb{P}_{RM(t)}^{1st}$ due to the fact that $x > y$. Therefore, the probability that $t(x, y)$ falls into this case is:

$$P(t(x, y) \text{ is spilled to the defer buffer} \mid x > y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - \sum_{i=y}^{x-1} \mathbb{N}_A(i) \\ 1 & \sum_{i=y}^{x-1} \mathbb{N}_A(i) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M} - \sum_{i=y}^{x-1} \mathbb{N}_A(i)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (8)$$

(III) With $x < y$, a tuple $t(x, y)$ cannot be in cases 2 and 3. Thus, $t(x, y)$ will not be spilled to the defer buffer, and $P(t(x, y) \text{ is spilled to the defer buffer} \mid x < y) = 0$. If $t(x, y)$ with $x < y$ is in case 4, then it has to be spilled to the hold buffer. Here, we have $\mathbb{P}_t \geq \mathbb{P}_{RM(t)}^{last}$. Similar to the previous case, the probability of $t(x, y)$ being in this case can be derived as follows:

$$P(t(x, y) \text{ is spilled to the hold buffer} \mid x < y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - \sum_{i=x+1}^y \mathbb{N}_A(i) \\ 1 & \sum_{i=x+1}^y \mathbb{N}_A(i) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M} - \sum_{i=x+1}^y \mathbb{N}_A(i)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (9)$$

Now we can derive the values of N_1 and N_2 as follows:

$$N_1 = \frac{\sum_{x,y} \mathbb{N}(x, y) P(t(x, y) \text{ is spilled to the hold buffer})}{\rho} \quad (10)$$

$$N_2 = \frac{\sum_{x,y} \mathbb{N}(x, y) P(t(x, y) \text{ is spilled to the defer buffer})}{\rho} \quad (11)$$

where ρ is the number of R tuples that can be stored in each page.

4.1.2 Cost with Prioritized Tuple Eviction Policy

The above analysis on the values of N_1 and N_2 does not consider tuple eviction priorities defined in Section 3.2.1, and hence may not reflect the real cost of our algorithm correctly. Below we try to measure some effects of our prioritized tuple eviction policy.

(I) A tuple $t(x, x)$ in case 4 now is more likely to be kept in the main buffer until all its right-matching tuples have been scanned. Therefore, $t(x, x)$ can meet with all of its matching tuples in the main buffer and can be directly discarded afterwards. As such, the probability of $t(x, x)$ being spilled into the hold buffer becomes:

$$P(t(x, y) \text{ is spilled to the hold buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ \frac{1}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (12)$$

By comparing with Eqn. (7), one can see that the adoption of tuple eviction priorities significantly reduces the probability of spilling $t(x, x)$ to the hold buffer. The number of tuples spilled to the defer buffer in this case is unchanged.

(II) Similarly, a tuple $t(x, y)$ with $x < y$ that belongs to case 4 is more likely to be kept in the main buffer until its right-matching tuples are fully scanned. Consider the set S of tuples in $S_A(R)$ whose attribute A values fall in the range $(x, y]$. Within S , those tuples in cases 1, 2 and 3 all have higher eviction priorities than $t(x, y)$. By assuming that $t(x, y)$ has a higher eviction priority than any case 4 tuple in S and that the total number of case 4 tuples in S is maximum, we can derive an upper bound of the probability that $t(x, y)$ is spilled to the hold buffer, which can serve as an approximation of the actual probability:

$$P(t(x, y) \text{ is spilled to the hold buffer} \mid x < y) \leq \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - k(x, y) \\ 1 & k(x, y) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M} - k(x, y)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \quad (13)$$

where

$$k(x, y) = \sum_{\substack{x < i \leq y \\ i \in j}} \mathbb{N}(i, j) \quad (14)$$

Again, by comparing with Eqn. (9), we can see that the prioritized tuple eviction policy can significantly reduce the probability of spilling $t(x, y)$ to the hold buffer.

4.1.3 Practical Considerations

In DBMS systems, we could utilize a two-dimensional histogram to summarize the joint distribution function $\mathbb{N}(x, y)$ and hence we can use the above cost model to estimate the cost of the join algorithm. Note that the sum aggregates in Eqns. (8) and (9) can be efficiently calculated with the widely adopted one dimensional equi-depth histogram and cumulative histogram. However, Eqn. (14) would be expensive to calculate. Therefore, we will adopt Eqn. (9) in our cost model for algorithm implementation, which provides an upper bound of the algorithm's cost.

When the two-dimensional histograms are unavailable, we will use the one-dimensional statistics to estimate $\mathbb{N}(x, y)$ as follows. Suppose we only have the one functions $\mathbb{N}_A(x)$ and $\mathbb{N}_B(y)$. By assuming that the attributes A and B are statistically independent of each other, we can derive the function $\mathbb{N}(x, y)$ as follows ($|R|$ is the total number of tuples in R): $\mathbb{N}(x, y) = \frac{\mathbb{N}_A(x) \cdot \mathbb{N}_B(y)}{|R|}$.

4.2 Comparison with Sort-Merge Join

Now we try to compare the cost of our **SCALE** algorithm with that of Sort-Merge Join (SMJ). The I/O cost of SMJ consists of: (a) the cost of externally sorting R into $\mathcal{S}_A(R)$, (b) the cost of externally sorting R into $\mathcal{S}_B(R)$ and (c) the cost of merge join of $\mathcal{S}_A(R)$ and $\mathcal{S}_B(R)$. Hence the total cost can be estimated as follows:

$$4N \lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 6N \quad (15)$$

In the worst case of **SCALE**, all the tuples will be spilled to the defer buffer during the first pass of processing $\mathcal{S}_A(R)$. That is, in this case, $N_2 = N$ and $N_1 = 0$. By substituting them into Eqn. (5), we have

$$2N \lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 2N \lceil \log_M \lceil \frac{N}{2M_f} \rceil \rceil + 6N$$

By comparing this with Eqn. (15), one can see the cost of **SCALE** would be the same as that of SMJ if $\lceil \log_M \lceil N/2M_f \rceil \rceil = \lceil \log_M \lceil N/2M \rceil \rceil$.

As described in Section 3.2.4, generally speaking, the larger the portion of tuples that are spilled to the defer buffer, the more memory is allocated to the hold and defer buffers. In the adverse case, M_f will be set to a value close to M such that $\lceil \log_M \lceil N/2M_f \rceil \rceil = \lceil \log_M \lceil N/2M \rceil \rceil$. In other words, **SCALE** degenerates to SMJ in the adverse case.

5. PERFORMANCE STUDY

We have integrated our proposed **SCALE** algorithm into PostgreSQL 8.4.4 [1] as a standard join operation. We enabled the query optimizer to additionally include **SCALE** in its plan search space, based on the cost model provided in Section 4. We used the default system settings without any tuning. We empirically compared **SCALE** with the native join operations of PostgreSQL: Sort-Merge Join (SMJ), Hybrid Hash Join (HHJ) and Nested-Loop Join (NLJ). However, the performance of NLJ was always significantly worse than the other three join operations in all experiments. Thus, we will not report the experimental results of NLJ here.

We conducted all experiments on a Dell workstation which is equipped with a Quad-Core Intel Xeon 2.66Hz CPU, 2GB DRAM and two SATA disks with storage capacities of 500GB and 750GB. Both the operating system, Ubuntu 7.10 with Linux 2.6.22 kernel, and the PostgreSQL system run on the 500GB disk, while the databases as well as intermediate results of PostgreSQL are stored on the 750GB disk.

5.1 Synthetic Dataset Generation

We generated numerous synthetic tables with different properties in order to comprehensively and extensively evaluate the performance of **SCALE**. In general, every synthetic table consists of two join attributes, A and B , along with another 23 padding attributes⁴. All attributes are of the (4-byte) integer data type and thus each tuple has a fixed size of 100 bytes. The attribute A , on which a synthetic table R will be externally sorted by **SCALE** to generate $\mathcal{S}_A(R)$, has the value domain $[1, 10^6]$.

As noted, the performance of **SCALE** is dependent on the overall distance (*nearness*) between tuples and their corresponding right-matching tuples in $\mathcal{S}_A(R)$. A shorter average distance means to us a stronger correlation between A

and B ⁵ and hence better performance of **SCALE**. We expect to test **SCALE** on synthetic tables with tunable correlation extents. To this end, we have four essential configurable parameters when generating a table R :

- **AD**: the statistical distribution of A values, uniform or Zipf.
- **MD**: the maximum absolute difference between the A value and the B value of a single tuple t , i.e., $|t.A - t.B| \leq \text{MD}$. MD is used to model the correlation between R.A and R.B. A small MD value means that R.A and R.B are more correlated. Hence, more tuples will be able to complete their right-joins before they are evicted from the main buffer. This is the situation where **SCALE** is expected to perform well. On the other hand, a large MD means that it is less likely for tuples to find matching tuples in the main buffer. Such cases are not favorable to **SCALE**.
- **DD**: the statistical distribution of $(t.A - t.B + \text{MD})$ values, either uniform or Zipf.
- **DV**: the number of distinct values on A , which are uniformly distributed over $[1, 10^6]$.

The Zipf distribution has a parameter θ , which affects the skewness of the data distribution: the greater the value of θ , the greater the skewness. We also varied the θ values. In the following presentation, we shall use “Zipf x ” to represent “Zipf with $\theta = x$ ”.

5.2 Experiment Design

On each synthetic table R , we executed self-join queries of the following basic form:

```
SELECT *
FROM R AS R1, R AS R2
WHERE R1.A = R2.B
```

and compared the total query execution times of **SCALE**, SMJ and HHJ. In certain queries, we also applied extra tuple selection conditions with different selectivities on both R_1 and R_2 . No clustered indices on A or B were available and thus both **SCALE** and SMJ were forced to explicitly sort R .

The experiments conducted consist of four parts. The first part is a micro-benchmark test, which enumerated different combinations of the above four parameters (AD, MD, DD and DV), as well as the total available join memory MEM on a set of synthetic tables with fixed sizes. The second part tested the scalability of **SCALE** by varying the table sizes. The third part measured and verified the effectiveness of our memory allocation scheme presented in Section 3.2.4. The final part focused on the performance of **SCALE** when combined with tuple selection and projection, as described in Section 3.3.

Note that in all experiments, during the first pass of **SCALE**, the size of the run buffer was set to MEM/10. Except for those experiments that studied the memory allocation scheme, for all other experiments, we completely relied on our memory allocation scheme to divide the remainder of MEM between the main buffer, and the hold and defer buffers. Between queries, we clear the operating system cache with the Linux command “`echo 3 > /proc/sys/vm/drop_caches`”.

⁵Note here the meaning of correlation is a bit different from its traditional definition, which measures the relationship between the A and B values within the same tuple.

⁴Note that with fewer padding attributes **SCALE** could perform even better, as the same amount of join memory now can hold more tuples and thus the sizes of the hold and defer buffers may decrease.

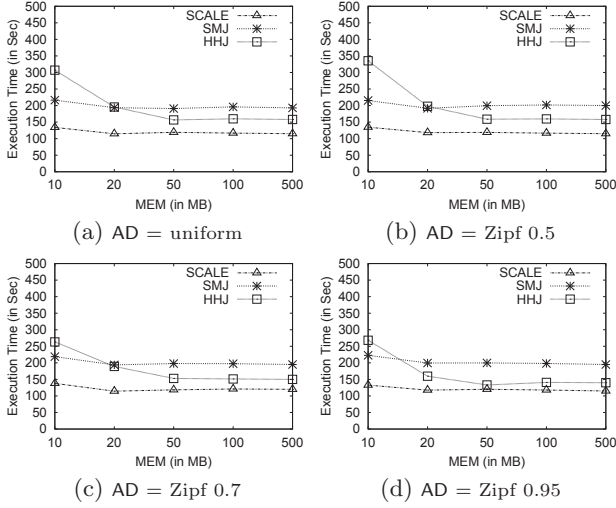


Figure 3: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5 , DD = uniform, DV = 1×10^5

We tested **SCALE** under a wide range of extents of correlation between A and B . Throughout our experiments (Fig. 3 to 10 below), we utilized three DV values, i.e. 10^5 (the most common), 5×10^5 and 9×10^5 , and two MD values, i.e. 10^5 (the most common) and 5×10^5 . In Fig. 3, 5, 8(a) and 10(a), DV = MD = 10^5 so that A and B were not obviously correlated; in Fig. 4, MD = 5×10^5 and DV = 10^5 so that A and B were much uncorrelated; in Fig. 6, 7, 8(b), 9 and 10(b), A and B were (a bit or very) correlated.

5.3 Experimental Results

5.3.1 Micro-Benchmark Test

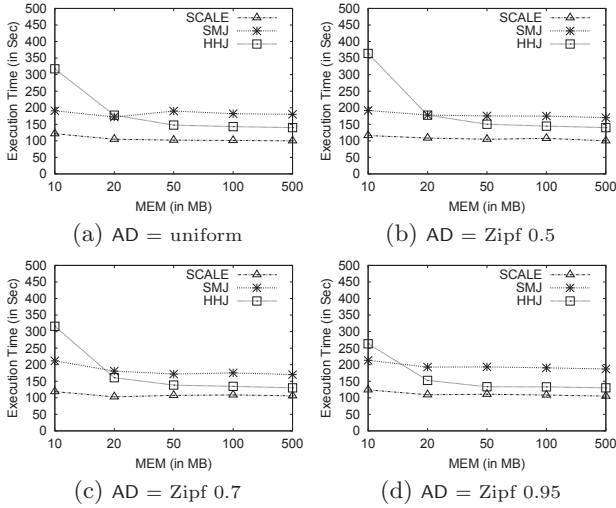


Figure 4: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 5×10^5 , DD = uniform, DV = 1×10^5

All synthetic tables in this test have 10 million tuples, with a total size of 1GB. The experimental results are depicted by Fig. 3 – 7, from which we can clearly see that **SCALE** sig-

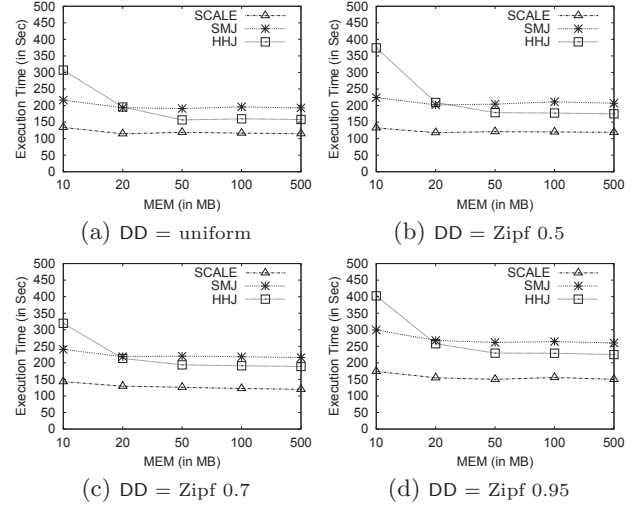


Figure 5: Benchmark test, 1GB tables with 10 million tuples, AD = uniform, MD = 10^5 , DD varies, DV = 1×10^5

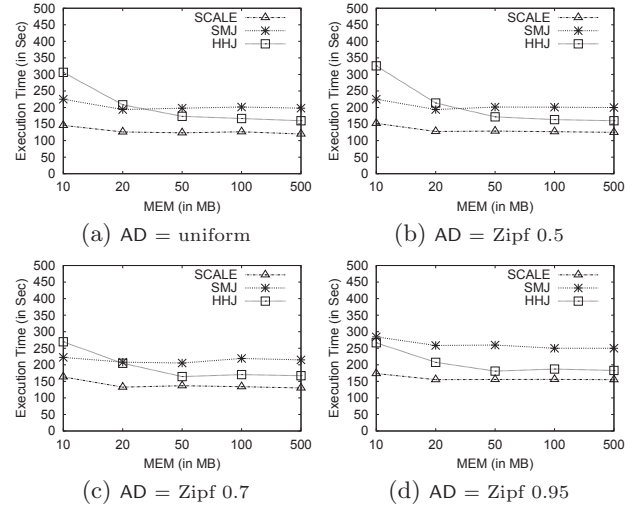


Figure 6: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5 , DD = uniform, DV = 5×10^5

nificantly outperformed both SMJ and HHJ in all situations. The performance gain of **SCALE** over the winner between SMJ and HHJ was between 20% to 45%. In all figures, we observe that the execution times of **SCALE** were quite stable for a wide range of join memory MEM. The execution times of SMJ and HHJ also stabilized when the join memory MEM increased to 100MB. We will not show the statistical details about tuple distribution over the six cases as well as the sizes of hold buffer tuples and defer buffer tuples in **SCALE**. Generally speaking, most tuples fell into cases 1, 2 and 4 as expected, among which case 1 tuples occupied a very significant portion. As a result, compared to the size of R , the total size of tuples in hold and defer buffers was usually small. The above observations explain the superiority of **SCALE**. We then briefly analyze the behaviour of **SCALE** according to the figures.

In both Fig. 3 and Fig. 4, when MEM was fixed, the execution times of **SCALE** remained nearly unchanged as AD

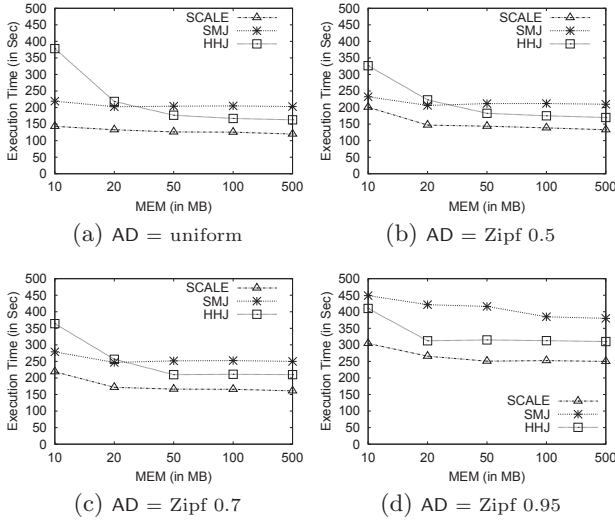


Figure 7: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5 , DD = uniform, DV = 9×10^5

varied. The underlying reason lies in that different AD settings resulted in more or less the same numbers of join result tuples, as well as the similar tuple distributions over the six cases. On the other hand, with a specific AD, when MEM increased, the number of tuples in case 2 (so is the size of defer buffer tuples) decreased slowly, the number of tuples in case 4 also decreased but the hold buffer was always empty. Therefore, the difference between the execution times of SCALE highly depended on how effectively the tuple runs in the defer buffer were merged. The fact is that multiple merge passes were required only when MEM = 10MB, which led to an execution time notably higher than those with larger MEM values. Comparing Fig. 3 with Fig. 4, the only parameter setting difference was the value of MD. With a greater MD, although the sizes of hold buffer tuples and defer buffer tuples increased, the number of join result tuples was reduced dramatically and thus much less CPU cost was incurred. Consequently, the execution times of SCALE in Fig. 4 were lower than their counterparts in Fig. 3.

In Fig. 5, the DD setting was varied. With the same MEM value, from Fig. 5(a) to Fig. 5(d), the sizes of hold buffer tuples and defer buffer tuples dropped gradually but the number of join result tuples rose quickly, and therefore the execution times increased correspondingly. Note that Fig. 5(a) is actually the same as Fig. 3(a). Within each of Fig. 5(b) – 5(d) having the Zipf DD, when the MEM increased, the size of defer buffer tuples decreased slightly while the hold buffer was always empty. Besides, multiple merge passes for tuples in the defer buffer were required only when MEM = 10MB. Therefore, in all four subfigures of Fig. 5, the execution times of SCALE with 10MB MEM were much higher than those with larger MEM values.

In both Fig. 6 and 7, when MEM was fixed and AD changed from uniform to Zipf (θ increasing from 0.5 to 0.95), both hold buffer tuples and defer buffer tuples shrank in sizes. However, in the meantime, the number of join result tuples increased, which incurred much more CPU time as well as a higher total execution time. On the other hand, with a specific AD, when MEM increased, the number of tuples in case 2 (so is the size of tuples in the defer buffer) decreased slowly, but the relatively small number of tuples in case 4 (so

is the size of tuples in the defer buffer) decreased fast. As a whole, similar to the scenarios in Fig. 3 and 4, the execution time differences of SCALE were determined by the number of merge passes when merging tuples in the defer buffer. Still, for the 10MB MEM, SCALE generated multiple merge passes and thus resulted in a higher execution time than others with larger MEM values. Among Fig. 3, Fig. 6 and Fig. 7, their parameter settings differed only on the value of DV. With a smaller DV, the sizes of hold buffer tuples and defer buffer tuples and the number of join result tuples all rose a bit. Consequently, the corresponding execution times of SCALE in Fig. 3, Fig. 6 and Fig. 7 were in an ascending order.

5.3.2 Scalability Test

In this test, we investigated how the relative performance of SCALE compared to SMJ and HHJ will change with respect to the synthetic table sizes. We fixed AD (uniform), MD (10^5) and DD (uniform), and then generated two groups of tables, each according to a different DV value (either 9×10^5 or 10^5). Each group contains four tables of 50 million (5GB), 100 million (10GB), 150 million (15GB) and 200 million (20GB) tuples, on which self-joins were conducted with join memory MEMs of 500MB, 1000MB, 1500MB and 2000MB respectively. The experimental results are plotted in Fig. 8.

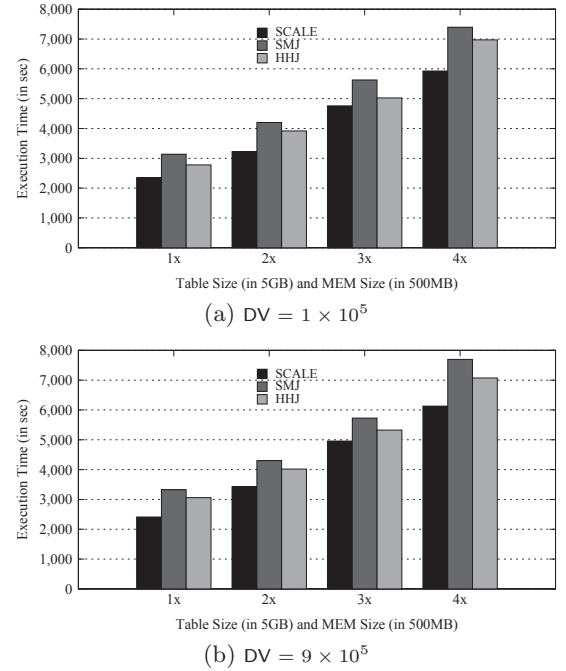


Figure 8: Scalability test, with varying table sizes and join memory sizes, AD = uniform, MD = 10^5 , DD = uniform

As shown, SCALE kept gaining significant performance improvement over both SMJ and HHJ as the table sizes increased. Moreover, all execution times of SCALE with DV = 9×10^5 in Fig. 8(b) were higher than their counterparts with DV = 1×10^5 in Fig. 8(a), which is consistent with our observations from Fig. 3, Fig. 6 and Fig. 7. As such, it would be convincing to claim that similar benchmark tests with different table sizes will bring the same conclusions on SCALE as those presented in the above micro-benchmark test.

5.3.3 Verification of Memory Allocation Scheme

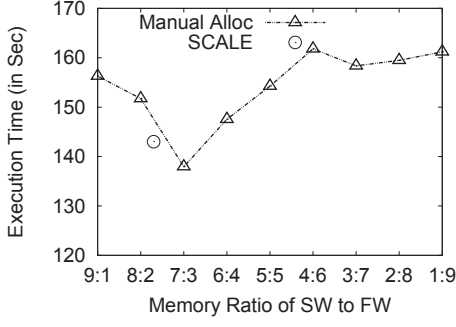


Figure 9: Verify the effect of memory allocation scheme, 1GB table with 10 million tuples, MEM = 10MB, AD = uniform, MD = 10^5 , DD = uniform, DV = 9×10^5

In order to verify the effectiveness of our memory allocation scheme proposed in Section 3.2.4, we conducted an experiment with the synthetic table in Fig. 7(a). We fixed MEM to 10MB and then ran SCALE with nine different memory ratios of the main buffer (denoted by SW) to the hold and defer buffers (denoted by FW). The experimental results are shown in Fig. 9.

It is obvious that the curve in Fig. 9 contains a trough whose lowest point corresponds to the ratio of 7:3 with the minimum execution time of 138 seconds. The small circle in Fig. 9 represents the chosen ratio, 77:23, by our automatic memory allocation scheme, with the actual execution time of 143 seconds. It turns out that our decision on the memory allocation is quite near to the optimal scenario in the exploited space.

5.3.4 Effect of Integration with Tuple Selection and Projection

It is desirable to see how the tuple selection and projection conditions that are pushed down to the joining instances of a self-join will affect the effectiveness and efficiency of SCALE. We therefore designed an experiment to investigate this.

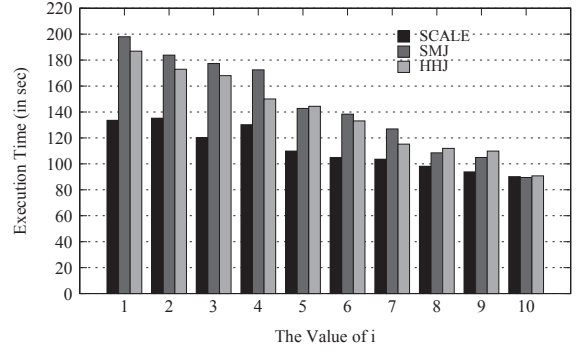
SCALE incorporated the first approach in Section 3.3 to enable tuple selection and projection pushdown, which requires much less implementation effort but has obviously worse performance than the second approach. We defined a refined self-join query template:

```
SELECT *
FROM R AS R1, R AS R2
WHERE R1.A = R2.B
      AND R1.C ≥  $i \times 5 \times 10^4$ 
      AND R2.C ≤  $10^6 - i \times 5 \times 10^4$ 
```

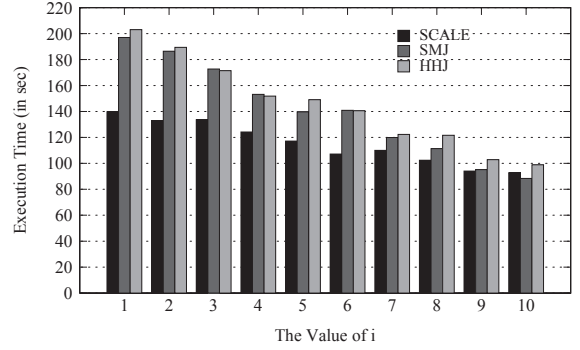
where C is a third integer attribute of R whose values are uniformly distributed over $[1, 10^6]$ and i is an integer parameter ranging from 1 to 10. By varying the value of i , we can easily and accurately control the tuple selection selectivities of R_1 and R_2 , as well as the number of overlapped tuples between these two instances. For simplicity we did not introduce tuple projection into the queries.

We tested the above refined self-join queries against two synthetic tables with fixed MEM (10MB), AD (uniform), MD (10^5) and DD (uniform) but two different DV values (1×10^5 and 5×10^5). The experimental results are shown in Fig. 10. In both SMJ and HHJ, the selection conditions in the queries were pushed down to the level of scanning R_1 and R_2 .

As can be seen, SCALE still performed better than SMJ and



(a) DV = 1×10^5



(b) DV = 5×10^5

Figure 10: Test on integration with selection condition $R_1.C \geq i \times 5 \times 10^4$ and $R_2.C \leq 10^6 - i \times 5 \times 10^4$, 1GB tables with 10 million tuples, MEM = 10MB, AD = uniform, MD = 10^5 , DD = uniform

HHJ in almost all scenarios. As i increased, the benefit of SCALE disappeared gradually. However, this trend is expected because the benefit of SCALE mainly originates from the overlap between R_1 and R_2 . When i became 10, which was actually the worst case as R_1 and R_2 are totally disjoint, the execution times of these three approaches are more or less the same. This phenomenon, however, is surprisingly positive. Note that regardless of the i value, in this test SCALE was always sorted on the original R and then two full sequential scans of $S_A(R)$ were conducted. Furthermore, as mentioned above, our implementation of SCALE chose the worse one of two candidate solutions for the purpose of combining tuple selection and projection. Therefore, we can optimistically conclude that a fully optimized SCALE will be superior to both SMJ and HHJ when dealing with general self-join queries.

6. EXTENSIONS TO SCALE

In this section, we propose two extensions to SCALE. The first extension can improve SCALE's performance, while the second extension generalizes SCALE for it to be utilized by more applications.

6.1 Sideways Information Passing

A tuple t in R plays roles as both the left-hand side (LHS) and the right-hand side (RHS) in the self-join condition $R_1.A = R_2.B$. Let $N_A(a)$ and $N_B(b)$ denote the total number of tuples in R that have the attribute value $A = a$ and $B = b$ respectively. Suppose SCALE sorts R on A into $S_A(R)$.

If $N_A(t.B)$ (resp. $N_B(t.A)$) is zero, t will not be able to

find corresponding right- (resp. left-) matching tuples in R . If both $N_A(t.B)$ and $N_B(t.A)$ are zero, then t is totally irrelevant to the self-join. Therefore, it would be beneficial to prune such irrelevant tuples from R as early as possible. To achieve this, we collect the value distribution information not only for attribute A , but also for attribute B , during the initial run formation phase of externally sorting R into $S_A(R)$. We can then discard those irrelevant tuples on-the-fly when merging the initial sorted runs during the subsequent run merge phase.

During the first pass of processing $S_A(R)$, it is safe and beneficial to remove a tuple t from the main buffer once t can no longer left-join or right-join with any other existing or incoming tuples in the main buffer. This is called *eager tuple pruning strategy*.

At the moment when t becomes eligible for the early pruning, $RM_S(t)$ must be empty. In the meantime, all the left-matching tuples of t must also have been read into the main buffer. This situation can be easily determined by counting the tuples whose attribute B values are equal to $t.A$ and so far have been read into the main buffer, and comparing the number with $N_B(t.A)$ which will be collected during the external sorting of R as described above.

6.2 Self Band-Join

A band-join [6, 13] between two relations R and S on attributes $R.A$ and $S.B$ has the join condition of the form $R.A - c_1 \leq S.B \leq R.A + c_2$, where c_1 and c_2 are constants that may be equal, and either one of them, but not both, may be zero. Band joins are common in queries that require joins over continuous domains such as time and distance. A self band-join involves two instances R_1 and R_2 of relation R with a join condition $R_1.A - c_1 \leq R_2.B \leq R_1.A + c_2$.

Extending **SCALE** to the self band-join is straightforward. For a tuple t with $t.B = b$, its right-matching tuples $RM(t)$ becomes the set of consecutive tuple segments in $S_A(R)$ with their A values falling into the range $[b - c_2, b + c_1]$. Besides, there are no other modifications required to enable **SCALE** to handle self band-joins.

The two schemes of sideways information passing discussed in Section 6.1 are also extendible according to self band-join. For a tuple t in R , when $\sum_{i=t.B-c_2}^{t.B+c_1} N_A(i) = 0$, t will not be able to find corresponding right-matching tuples in R ; similarly, when $\sum_{i=t.A-c_1}^{t.A+c_2} N_B(i) = 0$, t will not be able to find corresponding left-matching tuples in R . Therefore, if $\sum_{i=t.B-c_2}^{t.B+c_1} N_A(i) = 0$ and $\sum_{i=t.A-c_1}^{t.A+c_2} N_B(i) = 0$, then t is irrelevant to the self band-join and can be pruned during the external sorting of R into $S_A(R)$.

The principle of the eager tuple pruning for the main buffer also applies to the self band-join. However, in order to determine if all the left-matching tuples of a tuple t have been read into the main buffer, it requires counting all the tuples whose attribute B values fall in the range $[t.A - c_1, t.A + c_2]$.

It is also obvious that the self band-join can be integrated with tuple selection and projection pushdown as described in Section 3.3, since the merge join between $S_A(R^+)$ and $S_B(R^-)$ is also well adaptive to band-join.

7. CONCLUSION

In this paper, we have proposed **SCALE**, an efficient self-join algorithm. Our extensive performance evaluation showed that that **SCALE** is generally superior to conventional join al-

gorithms like Sort-Merge Join, Hybrid Hash Join and Nested-Loop Join. There are still several extensions and optimizations that potentially deserve to be studied in the future. First, in **SCALE**, the join result tuples are not delivered in a sorted order. However, in some cases, it would be desirable and beneficial to produce a sorted join output. It remains a challenge to revise the algorithm so as to generate sorted results without incurring too much overhead. Second, the memory allocation among buffers of **SCALE** is done statically based on the cost estimation before the first scan of the sorted relation. As the cost estimation might not be accurate due to inaccurate statistics, it would be interesting to design a dynamic allocation algorithm to adjust the memory allocation among all buffers at runtime. Third, **SCALE** does not exploit the opportunity of outputting join results during the external sorting of the relation at the beginning. Intuitively, interleaving the sorting with the tuple matching procedure would further improve the execution time. Fourth, our current techniques work well for a binary self-join. When executing a query with multi-way self-joins, there might be a more efficient way than simply applying a binary tree of **SCALE** operations.

Acknowledgements This research is supported in part by NUS Grant R-252-000-271-112.

8. REFERENCES

- [1] Postgresql. <http://www.postgresql.org/>.
- [2] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, 1999.
- [3] M. Blasgen and K. Eswaran. On the evaluation of queries in a relational database system. *IBM Res. Rep. RJ*, 1745, 1976.
- [4] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *VLDB*, 1997.
- [5] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2), 1984.
- [6] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equi-join algorithms. In *VLDB*, 1991.
- [7] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine GRACE. In *VLDB*, 1986.
- [8] C. A. Galindo-legaria, G. Graefe, M. M. Joshi, and R. T. Bunker. System and method for segmented evaluation of database queries. *US Patent No. 7,599,953*, 29 Nov. 2004.
- [9] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [10] G. Graefe. A generalized join algorithm. In *BTW*, pages 267–286, 2011.
- [11] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationships. In *VLDB*, 1998.
- [12] H. Lei and K. A. Ross. Faster joins, self-joins and multi-way joins using join indices. *Data Knowl. Eng.*, 29(2), 1999.
- [13] H. Lu and K. Tan. On sort-merge algorithm for band joins. *IEEE Transactions on Knowledge and Data Engineering*, pages 508–510, 1995.
- [14] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, 1988.
- [15] T. Pitoura and P. Triantafillou. Self-join size estimation in large-scale distributed data systems. In *ICDE*, 2008.
- [16] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.