

Lazy, Adaptive RID-List Intersection, and Its Application to Index Anding

Vijayshankar Raman¹

Lin Qiao¹

Wei Han¹

Inderpal S. Narang¹

Ying-Lin Chen²

Kou-Horng Yang²

Fen-Lin Ling²

¹IBM Almaden Research Center
San Jose, CA, U.S.A.

{ravijay,lsqiao,narang}@us.ibm.com

²IBM Silicon Valley Lab
San Jose, CA, U.S.A.

{ylchen,ayang,fenling}@us.ibm.com

ABSTRACT

RID-List (row id list) intersection is a common strategy in query processing, used in star joins, column stores, and even search engines. To apply a conjunction of predicates on a table, a query processor does index lookups to form sorted RID-lists (or bitmap) of the rows matching each predicate, then intersects the RID-lists via an AND-tree, and finally fetches the corresponding rows to apply any residual predicates and aggregates.

This process can be expensive when the RID-lists are large. Furthermore, the performance is sensitive to the order in which RID-lists are intersected together, and to treating the right predicates as residuals. If the optimizer chooses a wrong order or a wrong residual, due to a poor cardinality estimate, the resulting plan can run orders of magnitude slower than expected.

We present a new algorithm for RID-list intersection that is both more efficient and more robust than this standard algorithm. First, we avoid forming the RID-lists up front, and instead form this lazily as part of the intersection. This reduces the associated IO and sort cost significantly, especially when the data distribution is skewed. It also ameliorates the problem of wrong residual table selection. Second, we do not intersect the RID-lists via an AND-tree, because this is vulnerable to cardinality mis-estimations. Instead, we use an adaptive set intersection algorithm that performs well even when the cardinality estimates are wrong.

We present detailed experiments of this algorithm on data with varying distributions to validate its efficiency and predictability.

Categories and Subject Descriptors:

H.2.4[Information Systems][Database Management]

General Terms: Algorithms, Measurement, Performance

Keywords: AND tree, Intersection, Star Join, Lazy, Sorting

1. INTRODUCTION

Intersection of lists of row ids (RID-lists) is one of the most useful operations in query processing. It can be used whenever there is logically one huge table upon which one wants to apply a conjunction of predicates, and a table scan is either expensive or impossi-

ble. The following examples illustrate the use of RID-list intersection in varied situations.

Star Joins in Warehouses. The following query analyzes coffee sales at cities in California by joining a fact table Orders with multiple dimensions.

EXAMPLE 1:

```
SELECT  S.city, SUM(O.quantity), COUNT(E.name)
FROM    orders O, customer C, store S,
        product P, employee E
WHERE   O.custId=C.id AND O.storeId=S.id AND
        O.prodId=P.id AND O.empId=E.id AND
        C.age = 65 AND S.state='CA' AND
        P.category='COFFEE' AND E.type='FULLTIME'
        AND O.year between 2000 and 2005
GROUP BY S.city
```

When Orders is large, it is prohibitively expensive to answer such a query via a table scan on Orders. Instead, most DBMSs would use indexes to compute lists of matching RIDs for each predicate: $O1 = \{\text{Orders.rid} \mid C.\text{age} = 65\}$, $O2 = \{\text{Orders.rid} \mid S.\text{state} = 'CA'\}$, etc.. For each predicate, the RIDs formed from index lookups are usually in several sorted lists. So they need to be merged to form one list sorted by RID. They then compute $I = O1 \cap O2 \cap \dots$, then fetch rows in Orders corresponding to RIDs in I , and finally compute the aggregate on these rows.

Keyword Search. As a third example, consider a query for (“query” and (“optimisation” or “optimization”)) against a search engine. It is typically processed as follows. First, each keyword is separately looked up in an (inverted list) index to find lists L_{query} , $L_{\text{optimisation}}$, and $L_{\text{optimization}}$ of matching document ids, and the second and third lists are merged into one sorted list. Second, the two remaining lists are intersected and sorted by some measure of rank. Third, the intersected list of ids is used to lookup into a table of document summaries to fetch the summaries of the matching documents, which are then displayed to the user.

Query Processing in Column Stores. Recently there has been a spurt of interest in column stores (e.g., [15]). These store schemas like the above as denormalized “universal relations”, decomposed into separate columns for category, state, age, quantity, and so on. A column store does not store a RID with these decomposed columns; the columns are all sorted by RID, so the RID for a value is indicated by its position in the column. To answer the query of Example 1, a column store like C-Store [15] will do separate scans on each column to identify values matching each predicate, infer the RID-list from the positions of these matching values, then intersect the RID-lists and fetch and aggregate the quantity

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.

SIGMOD '07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

values corresponding to these RIDs (from the quantity column).

In the first two cases, the table is too large to process by a table scan. In the last case, the table is stored decomposed into columns, so RID-list intersection is the only option.

Currently, the most popular way of doing RID-list intersection is *index anding*: using a suitable index (often a bitmap index [11, 12]), construct one or several lists of matching RIDs for each predicate, merge all the RID lists for each predicate, and then intersect all the lists together to compute the intersection. This process is shown in Figure 1. The problem with usual implementation of index anding is: (a) the RID list merge is often expensive, and (b) the intersection is done via a binary AND-tree, whose performance is highly sensitive to the ordering of lists in the tree.

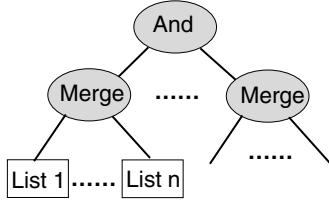


Figure 1: Index Anding

In this paper, we present a new RID-list intersection algorithm that addresses the above issues and improves on index anding in two ways: (a) we form and merge the RID lists lazily, only when necessary, and (b) we AND the RID lists adaptively. Hence, it is more robust in intersecting them, when cardinalities are mis-estimated. We analyse the extent of this robustness through competitive analysis, where we compare our worst-case performance against that of a perfect oracle.

We have implemented our algorithm as an intersection library in a prototype version of DB2 for z/OS. We present a detailed performance study of this prototype on data with uniform, skewed, and correlated distributions to quantify the efficiency and robustness of our algorithm. Before introducing this algorithm, we briefly review how index anding works.

1.1 Primer on Index Anding

We explain index anding using the query of Example 1. Figure 2 shows an execution plan for this query. It has four phases:

1. **RID-list formation:** The first step is to find RID-lists of Orders tuples that satisfy each predicate: (we will also use the term *leg* to mean the RID-list corresponding to a predicate):
 - We start by finding the matching rows in the dimension tables, and use the dimension keys in the matches to look up an appropriate foreign key index on Orders. The results of this lookup are lists of matching RIDs for each value of dimension key. We call these lists as *segments*.
 - We now have a collection of segments that satisfy the predicate (one for each value of the foreign key). We merge the RIDs in all these segments together to form a single sorted leg for this predicate.
E.g., for a predicate `category="coffee"`, we look up into an index on `O.prodId` with each value in `{P.id | category="coffee"}`.
2. **Intersection:** Compute the intersection of these legs via a tree of AND operators.
3. **Fetch:** The list of RIDs in the intersection, i.e., ones that match every predicate, is usually much smaller than the fact table size. So we directly fetch the fact table rows corresponding to these RIDs (RIDs are usually physical (page,slot) identifiers).

4. **Residuals:** Finally, join these fetched rows with all the remaining dimension tables (*residual tables*), to get any remaining columns and evaluate any residual predicates. With most predicates, the optimizer has a choice of applying them via the intersection or as a residual. Applying via the intersection has the benefit of early filtering of rows, but the cost is that of forming the matching RID-list. Usually only very poorly selective predicates (where almost no row is filtered) are chosen as residuals.

Although index anding is widely used, it has two problems:

- **Performance:** The join needs to form a segment of matching RIDs for each foreign key satisfying a predicate, and then merge these segments into a unified RID-list for each predicate. The segment formation involves scanning and parsing index leaf pages, and the merge involves sorting. Both of these operations are expensive when the number of matching RIDs is large.
- **Predictability.** The performance of index anding is very sensitive to two optimizer decisions:
 - Operator order in the AND-tree: Since the cost of a pairwise AND is related to the input sizes, it is important to AND early those pairs of legs that have small intersections. Figure 3 shows an expensive AND-tree where very large legs of sizes upto 20M are intersected first. Instead, small legs, such as P and C, should be intersected first.
 - Choice of residuals: Segment formation and merging is especially slow for poorly selective predicates that have lots of matching RIDs. This situation can arise if the optimizer does a bad job of choosing residual predicates. For example, Figure 4 shows another AND-tree for the same query where the predicate on Employee is applied during index-anding rather than as a residual. The cost of forming a merged leg of size 0.5B will alone be far higher than the cost of the entire plan of Figure 3.

If the optimizer mis-estimates cardinalities of one or more intersection legs, or due to correlation, mis-estimates the cardinality of an intermediate intersection result, it could pick a plan with a poor AND-tree or residual.

1.2 Adaptive Intersection and Lazy Merging

It is now well understood that cardinality estimation is a fundamentally hard problem, especially in the presence of correlations [14, 7]. To tackle both the efficiency and robustness shortcomings of index-anding, we propose a new algorithm for index anding that changes both the way RID-lists are formed for each predicate, and the way they are intersected.

- **Lazy Merging:** We do not form the full RID-lists for each value of foreign key up front. Instead, we do this on demand, during the intersection phase itself, thereby forming only the portion of the RID-lists needed for intersection. E.g., we only scan leaf pages that can possibly contain RIDs in the intersection result. Due to this deferred segment formation, we also need to defer the merging of segments to form sorted legs. But this can get expensive when one leg has several segments – during intersection, when we need to check if a leg has a RID, we must separately check in each of its segments. We propose an online algorithm that monitors this overhead and dynamically determines how long it is worthwhile to defer segment formation and merging. We prove that this algorithm is competitive to a factor 2 to an oracle that chooses the best time to merge each segment.
- **Adaptive Intersection:** We avoid the risk of cardinality mis-estimation by using an n-ary intersection rather than an AND-tree for intersecting the legs. This n-ary intersection is based on Demaine et al’s adaptive set intersection algorithm [6]. But unlike the round-robin policy used in [6], we dynamically vary which leg to look at based on the recent density of RIDs in each leg.

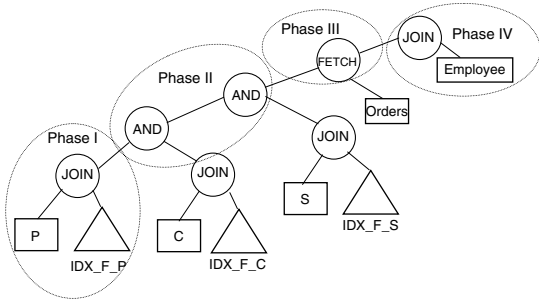


Figure 2: Example Index Anding Plan.
 $P = \pi_{id \sigma_{category='COFFEE'}}(product)$,
 $C = \pi_{id \sigma_{age=65}}(customer)$, $S = \pi_{id \sigma_{state='CA'}}(store)$.
 The JOIN operator in Phase I takes a list of keys, joins them with the index, and produces a sorted RID list. The AND operator takes two RID lists and intersect them.

1.3 Outline of the Paper

The next two sections describe the details of adaptive intersection and lazy merging. We then present an experimental evaluation in Section 4. We summarize related work in Section 5 and conclude with pointers to future work in Section 6.

2. ADAPTIVE INTERSECTION

We start with the intersection phase. For this section, we assume that for each predicate we have already formed a sorted list of RIDs matching that predicate. This list is exposed to the intersection via a *SortedRidList* interface:

- *RID start()*: opens a cursor on this sorted list
return the first RID
- *RID getNext()*: return the next RID and advances the cursor
- *bool eof()*: return true if cursor is at end of list
- *RID lookup(RID x)*: checks if x exists in this list:
if found, return x
else return the smallest RID $> x$

This interface is a simple list iterator, plus a lookup function that does a binary search. In Section 3 we describe how to support such an interface lazily, without actually forming a sorted leg up front.

2.1 Traditional AND-Tree

In terms of this *SortedRidList* interface, the AND-tree approach of Figures 1, 2, and 3 is simple. We construct a tree of iterators, where each iterator repeatedly calls *getNext()* on its left leg and *lookup()* on its right leg. For example, the following logic intersects two legs L_1, L_2 by an AND-tree (L_1, L_2) :

1. for each RID x in L_1 do
2. $y \leftarrow L_2.lookup(x)$
3. if $(y = x)$ output y

The following notation is helpful in analysing the sequence of *lookups* that occur during intersection. Consider each RID as a point in a number line $[0, |fact\ table|]$. So a leg of sorted RIDs becomes a number line annotated with hash marks corresponding to its elements. For example, in Figure 5, the densely marked horizontal line L_1 is a leg with many RIDs, while the sparsely marked line L_2 is a leg with few RIDs.

Figure 5a1 illustrates the above AND-tree: each *lookup()* is shown by an arrow: successful matches are solid arrows and unsuccessful

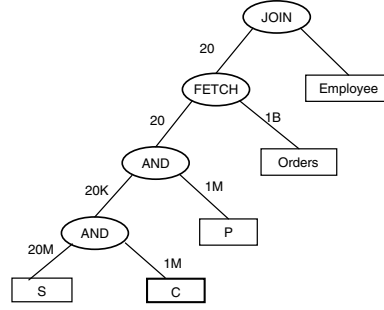


Figure 3: Poor And-tree and Right Residuals

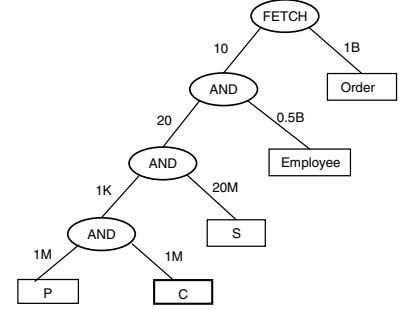


Figure 4: Right And-tree and Poor Residuals

ones are dashed. One could also do the intersection with a converse AND-tree (L_2, L_1) (Figure 5a2), where we scan RIDs in L_2 and lookup into L_1 .

If the leg sizes are $m = |L_1|$ and $n = |L_2|$, the first AND-tree takes $O(m \log n)$ time and the second takes $O(n \log m)$ time. If the leg sizes are very different (say 10^3 and 10^6), a wrong AND-tree, which could be chosen due to bad cardinality estimation, will perform much worse than the right one.

We now present an alternative way to do intersections that avoids this risk of choosing a wrong AND-tree. We start with a round-robin intersection algorithm proposed by Demaine et. al [6] before describing our probabilistic algorithm.

2.2 Round-Robin Intersection

Consider the following variant of the (L_1, L_2) AND-tree.

1. $x \leftarrow L_1.start()$
2. while not $(L_1.eof())$ and $L_2.eof()$ do
3. $y \leftarrow L_2.lookup(x)$
4. if $(y = x)$ {output y ; $y \leftarrow L_2.getNext()$ }
5. $x \leftarrow L_1.lookup(y)$
6. if $(x = y)$ {output x ; $x \leftarrow L_1.getNext()$ }

The essential idea, as illustrated in Figure 5b is to sweep through the number line of all RIDs by *alternate lookup()*s into L_1 and L_2 . Initially, the entire range $[x = 0, \infty)$ could be in the intersection. If $L_2.lookup(x)$ returns y which is $> x$, we know for sure that none of RIDs in $[x, y)$ belongs to the intersection. So we continue the sweep in $[y, \infty)$.

The robustness of this alternating algorithm is easily seen from Figure 5b. Every RID in the smaller list L_2 is incident to at most two arrows: so there are at most $2 \min(m, n)$ lookup()s. In practice, the performance is even better, because *lookup()* can jump over several RIDs.

The algorithm proposed by Demaine et al is a n -ary version of this 2-ary alternating intersection. The algorithm still sweeps through the domain of RIDs by repeatedly invoking *lookup()*. But, instead of switching back and forth between two lists L_1 and L_2 , this algorithm does *lookup()* into all the n lists being intersected, in a round-robin fashion. Only when a RID has found matches in all the n lists does it output the RID and invoke *getNext()* to move on to the next RID. Otherwise, it continues to invoke *lookup()*.

Figure 6c illustrates this process in an intersection of 4 legs. No-

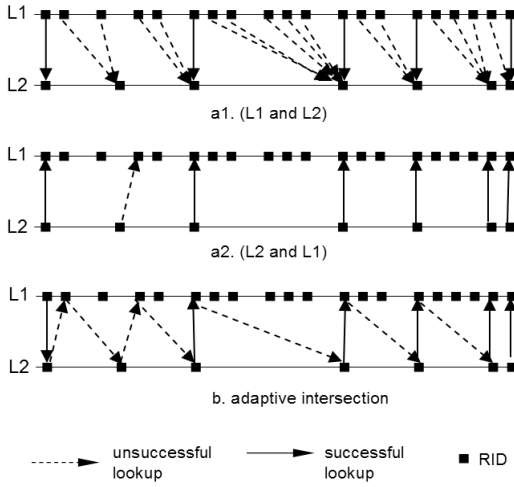


Figure 5: Three ways of intersecting two sorted lists

tice that we start on the dense legs L_1, L_2 , but because of the round-robin nature, the number of lookup(s) is proportional to the size of the sparsest leg L_4 .

In contrast, an AND-tree like $((L_1, L_2), L_3), L_4$ (Figure 6a) performs a huge number of lookup(s) intersecting the dense legs L_1 and L_2 , and so performs far worse. The best AND-tree for this intersection is $((L_4, L_3), L_2), L_1$, shown in Figure 6b.

This round-robin intersection has been extensively analyzed in [6] and [3], and it is known to always perform within a factor k of the best AND-tree, where there are k legs. Intuitively, a round-robin intersection looks up in every leg, and in particular, in the most sparse leg, once in k lookups. So it will never perform more than k times as many lookups as the best AND-tree,

2.3 Probabilistic Intersection

As compared to an AND-tree, this round robin intersection is robust. But it is rather conservative, and a correctly chosen AND-tree can perform better. In the example of Figure 6c itself, the round robin persistently does lookup(s) into lists L_1 and L_2 , even though they provide no benefit: the lookup(s) into L_3 and L_4 are the only lookup(s) that skip any RIDs at all. An AND-tree that first intersects L_3 and L_4 and then intersects the result with L_1 and L_2 will do almost three times fewer lookup(s). In general, when there are n legs, round robin intersection can do n times more lookups than the best AND-tree that a perfect optimizer would choose.

To avoid this slowdown, we propose an alternative probabilistic intersection. The idea is to maintain, for each leg L_i , a historical average of $L_i.\text{lookup}(x) - x$: the average number of RIDs skipped in lookup(s) to that leg in the past. We call this the *average jump* for leg L_i .

Instead of doing lookup(s) always in round robin fashion, we choose the next leg to lookup into in one of two ways (Figure 7). Suppose that the most recent lookup(s) was into leg L_j :

- *Lookup into the most promising legs:* With probability p , we do the next lookup into the leg with the highest average jump in the recent past. If this leg is L_j itself, there is no point in doing another lookup(s) into it now – we will obviously find a match. Instead we do a lookup into the leg with the second highest average jump.
- *Lookup into an arbitrary leg:* With probability $1 - p$, we do the next lookup in round robin fashion, into $L_{j+1\%(\# \text{ legs})}$. This method is similar to the explore-exploit tradeoff used in the

```
// Gets the next Leg to search, probabilistically
nextList(p)
1.  with probability p go round-robin;
2.    return (currLeg+1) % number of legs;
3.  with probability 1 - p go to most promising leg;
4.    if currLeg!=bestLeg
5.      return bestLeg;
6.    else if currLeg!=nextBestLeg
7.      return nextBestLeg;
8.    else return (currLeg+1) % number of legs;
```

Figure 7: Pseudocode for probabilistic intersection.

machine learning literature. The first option exploits the information about the best legs (least density of RIDs) we have seen so far. The second option explores other legs to learn if their RID density has changed.

In an example like Figure 6a, the average jump for L_3 and L_4 will quickly rise to be much more than that for L_1 and L_2 , so this adaptive method will perform almost as well as the best AND-tree.

If a predicate is correlated with RID, the density in the corresponding leg will vary as the intersection progresses. Any single AND-tree will perform well in some regions and poorly in others. This probabilistic intersection on the other hand will adapt to the changing RID densities by switching to the less dense legs. As a result, probabilistic intersection can outperform even the best statically chosen AND-tree. In order to track changes in the average jump sizes (*i.e.*, RID density) in a leg, we decay the average jump periodically by a factor of two.

Our method has two parameters: the decay rate and the probability p . Setting the probability low will force the intersection to focus on the most selective legs without wasting time on other legs. However, an index leg might become less selective over time due to skew, and a low probability value may prevent the algorithm from learning the selectivity change in time. Setting the probability high will allow for faster learning but has higher overhead in exploring unselective legs. We have experimented with a variety of settings and found that $p = 0.25$ gives a good balance between performance and robustness. We currently decay the average jump by half every 100 lookup(s); we have found this to work well for a variety of data distributions that we study in Section 4.

3. LAZY MERGING

Having seen how to make the intersection phase perform more predictably, we now turn to the merge phase. The purpose of this phase is to realize the SortedRidList abstraction we used during intersection, in particular, the getNext() and lookup() primitives.

Eager Merging

The standard way of realizing this interface to merge the RID-lists on each leg before the intersection. We will call this method as *Eager Merging*. For example, consider the predicate $\text{store.country} = \text{UK}$. Let the dimension keys satisfying this predicate be: $\pi_{\text{storeId}}(\sigma_{\text{store.country}=\text{UK}} \text{Store}) = \{k_1, k_2, \dots, k_p\}$. Eager Merging first looks up into the index on Orders.storeId with each dimension key k_i to form a *segment* of matching rids: $L_i = \{\text{Orders.rid} | \text{Orders.storeId} = k_i\}$. Then it merges these segments together, to form a single sorted RID-list: $L = L_1 \cup L_2 \cup \dots \cup L_p$.

We next describe two optimizations that make this process more efficient. We start with an optimization that postpones the segment merging (assuming that every segment has been formed by read-

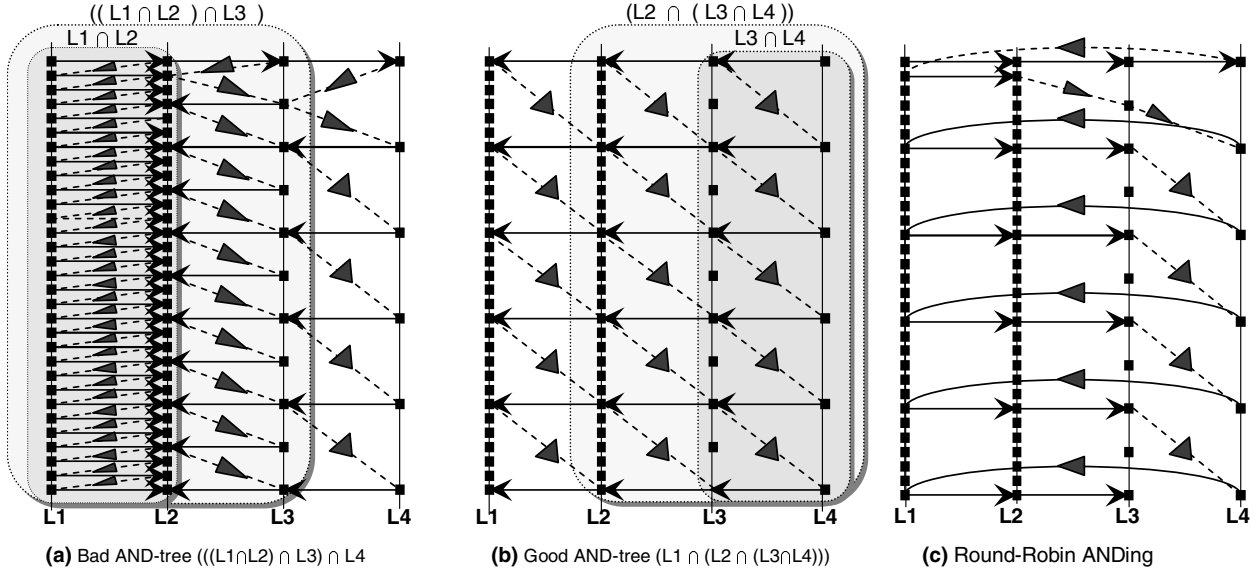


Figure 6: 3 ways of intersecting 4 RID-lists. Lists are shown as vertical lines with RIDs (in sorted order) shown as hash marks. Intersection algorithm is illustrated with solid arrows (successful lookup) and dashed arrows (unsuccessful lookup).

ing in the leaf pages). We then present a further optimization that avoids reading in the segments upfront, and lazily reads in only those portions of each segment that are needed for the intersection.

These optimizations exploit two properties of B-Tree indexes:

- The RIDs within each segment are sorted by RID. *i.e.*, in an index, all entries for a given key are ordered by RID. This property holds true for B-Tree indexes, as long as there are no in-place updates. Newly inserted records get higher RIDs than previous records so they don't affect the sort order. Deleted records are marked as tombstones, and their RIDs are reused only upon a reorganization. The policy of doing no in-place update for fact tables is something followed in most warehouse implementations – updates are rare, and are handled as insert/delete pairs.
- In the lowest level non-leaf pages, each child (leaf page) pointer contains the highest RID in that leaf page. Essentially, this means that a single-index on a column c is treated as an index on (c, RID) .

Note that both of these properties would implicitly hold if the segments were formed by lookups into a bitmap index. First, a segment stored as a bitmap represents its RIDs by turning on bits at appropriate positions in a bitmap. Hence RIDs in a bitmap are available in sorted order. Second, if a segment stored as a bitmap spans multiple pages, each page has a constant number of bits. So the highest RID in a page can be well estimated by using the position of that page in the bitmap. Thus, our optimizations would hold for bitmap indexes as well.

3.1 No Merging and Lazy Merging

A simple alternative to Eager Merging is *No Merging*: directly intersect unions of unmerged segments. The SortedRidList interface used in intersection can be implemented over unmerged segments $L_1 \dots L_p$ of a leg L thus:

- `start()`:
open a cursor on each $L_i \mid 1 \leq i \leq p$
- `lookup(rid)`:
return $\min_{0 \leq i \leq p} L_i.lookup(rid)$.
- `getNext()`:
let $currSeg \leftarrow \operatorname{argmin}_{0 \leq i \leq p} L_i.currentRid()$.
return $L_{currSeg}.getNext()$.

- `currentRid()`:

return $\min_{1 \leq i \leq p} L_i.currentRid()$.

Observe that each operation on an unmerged leg L does p operations of the same kind, one on each of the unmerged segments of L . In effect, intersection with a union of unmerged segments is equivalent to pushing the intersection below the union via distributive property. For example, consider three legs that need to be intersected: A with 2 segments, B with p , and C with q . Eager Merging computes the intersection as

$$((A_1 \cup A_2) \cap (B_1 \cup B_2 \cup \dots \cup B_p) \cap (C_1 \cup C_2 \cup C_3 \cup \dots \cup C_q))$$

whereas No Merging computes the intersection as

$$((A_1 \cap B_1 \cap C_1) \cup (A_1 \cap B_1 \cap C_2) \cup \dots \cup (A_2 \cap B_p \cap C_q)).$$

With No Merging, the gain from avoiding the merge is offset by the increased cost of lookup() – we need to do separate lookup()s in each unmerged segment. Superficially, it would seem that No Merging pays off for legs with a small number of large segments, such as leg A , but not for legs with large number of small segments, like leg B . However, the cost of No Merging a leg is also determined by the number of lookup()s that happen on that leg during intersection: each lookup() on B is p times more expensive than if we had merged it up front, but the intersection may not invoke that many lookup()s on B at all. Moreover, even within the same leg we want to differentiate between very large and very small segments.

We quantify the tradeoff of merging a segment L_i in leg L thus:

$$cost_{eager}(L_i) = c_1 \times |L_i| \quad (1)$$

$$cost_{nomerger}(L_i) = c_2 \times \text{numLookups}(L) \quad (2)$$

where $\text{numLookups}(L)$ is the number of times the intersection logic invoked lookup()s on leg L . (1) follows from a linear cost model for a merge of sorted lists; we have attributed to L_i its contribution to the total merge cost (we ignore the slow growing $\log \# \text{segments}$ factor to simplify the analysis). (2) is because if we don't merge L_i , every lookup() on leg L has to do an extra lookup() on L_i .

Lazy Merging

We do not know $\text{numLookups}(L)$ beforehand. It depends loosely on the number of RIDs in the intersection of all the other legs (because the intersection logic has to check every one of them for ex-

istence in L). But, rather than trying to estimate $numLookups(L)$ from selectivities, we use an online algorithm similar to the well-known one for the rent-vs-buy problem. We start with no segments merged, and merge segments lazily, merging L_i after $\alpha|L_i|$ lookups:

1. Initially, leave all segments in all legs un-merged.
2. For each leg L with segments $L_1 \dots L_p$, add an extra segment L_0 to hold the segments to be merged in future:
 $L_0 \leftarrow \Phi$
3. During intersection, implement the SortedRidList interface on leg L as follow:
`lookup(rid):`
 $numLookups(L)++$
 if ($numLookups(L) \geq \alpha|L_i|$) do $L_0 \leftarrow merge(L_0, L_i)$.
 return $\min_{1 \leq i \leq p} L_i.lookup(rid)$.
`getNext():`
 $currSeg \leftarrow argmin_{1 \leq i \leq p} L_i.currentRid()$.
 return $L_{currSeg}.getNext()$.
`currentRid():`
 return $\min_{1 \leq i \leq p} L_i.currentRid()$.

Competitive Analysis

The cost of Lazy Merging depends on whether there are at least $\alpha|L_i|$ lookups or not:

$$cost_{lazy}(L_i) = |L_i|c_1 + \alpha|L_i|c_2 \text{ if } numLookups(L) \geq \alpha|L_i| \\ = numLookups(L)c_2 \text{ else.} \quad (3)$$

We now show that this is within a factor of 2 of an oracle that follows the best possible merge policy. This oracle knows apriori the value of $numLookups(L)$ for each leg L , and is allowed to choose Eager Merging or No Merging for each segment depending on which is faster, as calculated from formulas (1) and (2).

THEOREM 1. *If we set $\alpha = c_1/c_2$, the total cost of Lazy Merging policy, including the cost of any merges and any extra lookup(s) into unmerged segments is $\leq 2 \times$ cost of the oracle merge policy.*

PROOF. For any segment L_i , from (3),

$$cost_{lazy}(L_i) \leq |L_i|c_1 + \alpha|L_i|c_2, \text{ and}$$

$$cost_{lazy}(L_i) \leq numLookups(L)c_2 + numLookups(L)c_1/\alpha.$$

Hence, from (1) and (2),

$$\frac{cost_{lazy}(L_i)}{cost_{eager}(L_i)} \leq \frac{(c_1|L_i| + c_2\alpha|L_i|)}{(c_1|L_i|)} = 1 + \frac{c_2\alpha}{c_1}$$

$$\frac{cost_{lazy}(L_i)}{cost_{nomerge}(L_i)} \leq \frac{(c_1 numLookups(L)/\alpha + c_2 numLookups(L))}{(c_2 numLookups(L))} \\ = 1 + c_1/(\alpha c_2).$$

The oracle achieves $\min(cost_{eager}(L_i), cost_{nomerge}(L_i))$. So

$$\frac{cost_{lazy}(L_i)}{cost_{oracle}(L_i)} \leq \max(1 + \frac{c_1}{\alpha c_2}, 1 + \frac{\alpha c_2}{c_1})$$

By setting $\alpha = c_1/c_2$, $cost_{lazy}(L_i) \leq 2 \times cost_{oracle}(L_i)$ $\square$.

Choosing Potential Constant

The value of $\alpha = c_1/c_2$ depends on the implementation costs of sorted list merge and lookup. We now present a micro-benchmark experiment to choose this constant.

We evaluate an intersection of 4 legs, DP, DC, DT, DS from our experiment schema (Section 4). We fix the selectivities of DP, DT, DS and vary that of DC from 0.01 to 0.25. Figure 8 plots the elapsed time of Lazy Merging for various values of α (we call this the potential constant).

Our merging policy is to merge a segment L_i when $numLookups(L) \geq \alpha|L_i|$. So $\alpha = 0$ corresponds to Eager Merging. Likewise, $\alpha = \infty$ corresponds to No Merging. Increasing values of α involve merging with increasing degree of laziness, implying greater lookup() cost and lower merge cost. Based on these results, we pick a value of $\alpha = 1$ as a sweet spot. We use this α in the experiment section.

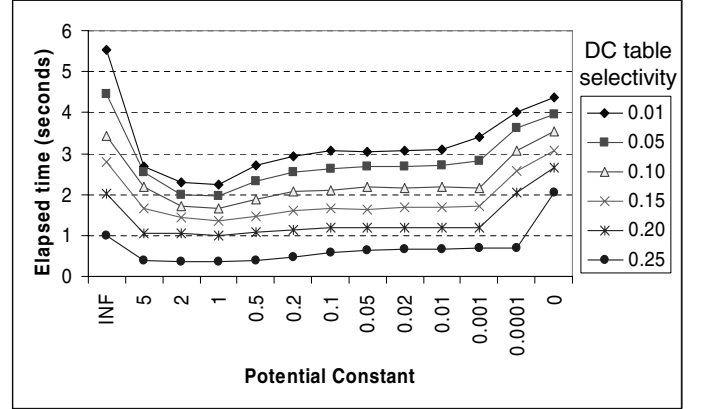


Figure 8: Performance of Lazy Merging for $DP \cap DC \cap DT \cap DS$ for various degrees of laziness. Selectivities on DC varies, on other legs it is: $DP = 0.02$, $DT = 0.1$, $DS = 0.5$.

3.2 Lazy Segment Formation

Up till now, we have assumed that the segments have been formed up front and have only deferred the merge. We now turn to lazy segment formation.

For a segment L_i in leg L , let P be the set of rids used as parameters in calls to $L.lookup()$. The only reason to form L_i is to perform these lookups, and any values of L_i that are not used in these lookups are irrelevant.

Consider an operation $L_i.lookup(r)$. This operation performs a search, returning r if it is found, and the immediately next higher rid in L_i otherwise. So there are only two values of L_i that affect the behavior of this lookup: r (if present), and $r' = \min\{x \in L_i \mid x > r\}$ (if r is found, the lookup is followed by a `getNext()` that fetches r').

We use this property to reduce the portion of L_i that we form, and to defer the formation until it is needed during intersection. Since L_i is a segment of a leg, the values in L_i are the RIDs corresponding to a single index lookup: e.g., $\{Sales.rid \mid Sales.storeId = 23\}$. So they correspond to the RIDs in a sequence of leaf pages of the index. For example, Figure 9 shows six leaf pages for a segment of leg M that is being intersected with legs L1, L2, and L3 (P is $L1 \cap L2 \cap L3$). As in the last section, segments are drawn as number lines with RIDs as hash marks. Notice that the rids in P correspond to only the first, third, and sixth leaf page in the sequence for L_i . Hence we can avoid forming L_i fully; we can skip the rids from the second, fourth, and fifth leaf page, and save the corresponding I/Os as well.

Recall that in a B-Tree, each child pointer of non-leaf pages at lowest level includes the highest RID among the leaves of that child. We use this highRid to decide, for a given rid in P , which leaf pages of L_i could be relevant. In the example of Figure 9, the highRid for the first page is 500 and that for the second is 1000. So the second page is irrelevant during a call for $M.lookup(1101)$.

This highRid information is usable in all but the last leaf page for a segment. The last leaf page could be shared with other segments

and so do not have a meaningful highRid. We only know that all rids in the last leaf page are $>$ highRid of the penultimate leaf page.

For a leg L , the set P of RIDs that are parameters in calls to $L.lookup(rid)$ is determined only during intersection. So, we form each segment L_i of L in a lazy fashion as follows:

- Before intersection begins, we do an index lookup to form a list of page numbers of the leaf pages that contain L_i , and an array $highRids(L_i)$ of the highest RID on each leaf page. Notice that this process involves only the non-leaf pages of the index, which are almost always cached in main memory. The only exception is when the entire segment is contained in a single leaf page – the highRids tell us nothing about the range of RIDs in this case, and so we scan the leaf page and extract the RIDs.
- During the intersection, L_i starts out unmerged with the other segments. Upon an invocation $L_i.lookup(rid)$ (via $L.lookup(rid)$), we do a binary search in $highRids(L_i)$ to find the leaf pages that could possibly contain rid or the immediately next higher rid, and scan these leaf pages alone: i.e., if $(highRids[i] < rid < highRids[i+1])$ we scan the $i+1$ 'st leaf page, and if $(highRids[i] = rid)$ we scan the i 'th and $i+1$ 'st leaves. $L_i.lookup(rid)$ can be evaluated using these pages alone. We cache the scanned pages so that we can reuse them if there is a subsequent invocation of lookup with a rid in the same range.
- During the intersection, when $numLookups(L) \geq \alpha|L_i|$, we need to merge L_i with other segments. At this point we scan all the leaf pages in L_i .

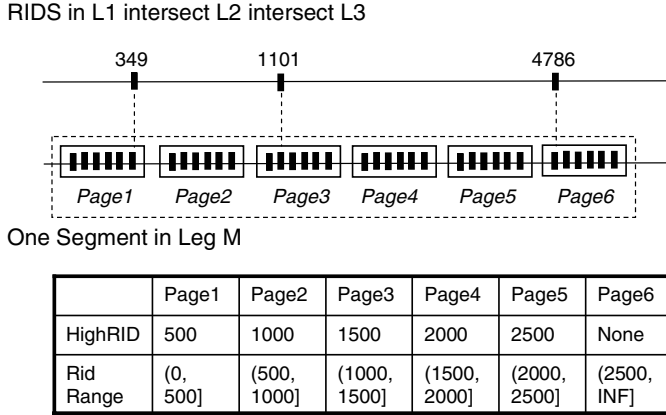


Figure 9: Avoiding Page Scan using HighRids

The success of lazy segment formation on a leg L depends on the distribution of RIDs in P . If the rids in P are uniformly spread throughout the domain of RIDs, it is likely that they will straddle many pages of L . Still, we can save if the number of RIDs in P is smaller than the number of leaf pages in L , which happens for many selective queries (note that P is the intersection of all legs other than L).

Another condition that facilitates lazy segment formation is correlation of intersection predicates with rid. This often happens with predicates on the time (or date) dimension. Fact table RIDs are invariably strongly correlated with time because transactions are dominated by inserts and deletes. Many queries restrict their interest to values in a specific date range, via a predicate. Hence, if L is a leg on any predicate other than date, its P will be correlated with RID. So it is likely that all the RIDs in P will match against only a subset of the leaf pages for each segment $L_i \in L$, and we can avoid scanning and parsing the other leaf pages.

4. EXPERIMENTAL EVALUATION

We have implemented our lazy merging and adaptive intersection algorithm as an intersection library that is invoked from a prototype version of DB2 for z/OS. We now present a detailed performance evaluation of this implementation. As in the previous sections, we divide our evaluation into two pieces:

- efficiency improvements by using lazy merging to form the sorted list of rids for each leg (Section 4.1), and
- predictability improvements by using adaptive intersection vs intersection via a statically chosen AND-tree for computing the intersection of the sorted lists (Section 4.2).

In each piece, we keep the other fixed: i.e., while evaluating lazy vs eager merging we use a statically chosen AND-tree (we choose the best AND-tree by hand), and while evaluating adaptive intersection we merge rids eagerly up front (so that the intersection is on sorted lists). At the end, in Section 4.3 we have a combined evaluation of lazy merging with adaptive intersection for completeness. Our experiments are run on an IBM RS/6000 pSeries 630 server with one Power 4 processor and 8GB of RAM.

4.1 Value of Lazy Merging

There are two opportunities for increased efficiency from lazy merging:

- we reduce the portion of each segment that has to be *scanned and parsed*, by combining with intersection and using the HighRid indicators in the B-Tree index.
- we reduce the amount of *merges* by delaying or avoiding the merge.

We expect to derive CPU savings from the latter and both IO and CPU savings from the former, with both contributing to reduce the elapsed time. We now quantify the extent of these savings under different data distributions.

We use a star schema based on a subset of tables in the SAP/BW benchmark [13]. It has 1 fact table and 7 dimension tables, as shown in Figure 10. The fact table *Orders* has a single column index for each of the seven foreign key columns: product_id, customer_id, etc. The cardinality column shows the number of primary keys on each table. Thus, *Orders* has 10 million records and the average number of rids per value in a foreign key column is (10M / cardinality).

Name	Cardinality	Clustered fact table index	Skewed Distribution
Orders (O)	10,000,000	-	-
Product (DP)	100,000	N	N
Customer (DC)	10,000	N	N
Time (DT)	1,000	Y	N
Employee (DE)	500	N	N
Account (DA)	1,000	N	Y
Store (DS)	10	N	N
Warehouse (DW)	2	N	N

Figure 10: Tables Used in Lazy Merging Experiments

Lazy segment merging is designed to do particularly well when the distribution has skew. With skew, some segments (foreign key values) will have many more RIDs than others, and lazy merging can exploit this by preferentially merging the small segments. Likewise, lazy segment formation pays off especially when a query predicate is correlated with the RID (recall discussion from Section 3.2). To test this, we start in Section 4.1.1 with a worst case of uniform distribution and no rid correlation, then move on to correlation (4.1.3) and skew (4.1.4). We also run one experiment where the optimizer chooses a poorly selective table for index and (one which should have been joined as a residual) in Section 4.1.2.

4.1.1 Uniform Distribution With No Rid Correlation

We start with a query over tables whose foreign key columns are uniformly distributed and uncorrelated with the RIDs in Orders:

```
SELECT DS.city, SUM(O.quantity), COUNT(DE.name)
FROM Orders O, Product DP, Customer DC,
Employee DE, Store DS
WHERE O.customer_id=DC.id AND
O.product_id=DP.id AND
O.store_id=DS.id AND
DC.age = <value> AND DP.category=<value> AND
DE.type= <value> AND DS.city=<value>
GROUP BY DS.city
```

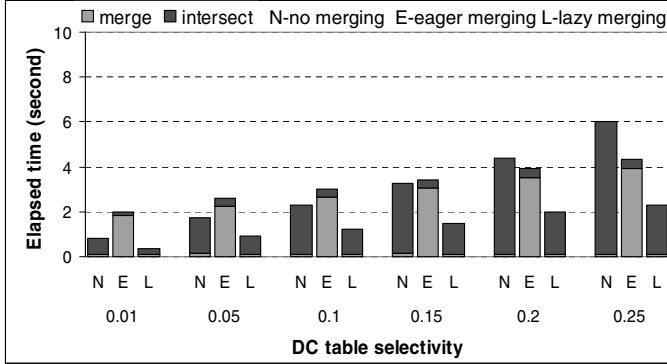


Figure 11: Lazy merging, Eager Merging, and No Merging for $DC \cap DP \cap DE \cap DS$. Leg selectivities on $DP = 0.02$, $DE = 0.1$, $DS = 0.5$, and that of DC varies from 0.01 to 0.25.

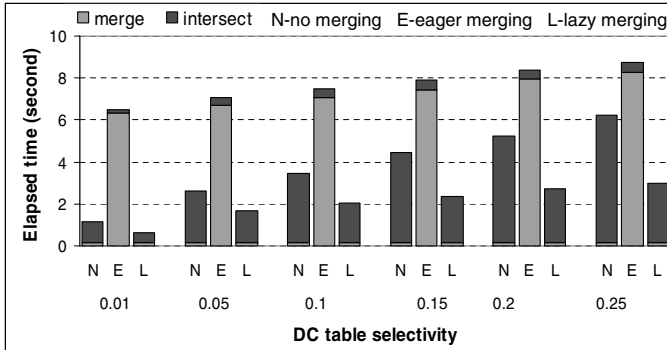


Figure 12: Lazy, Eager, and No Merging for $DC \cap DP \cap DE \cap DS \cap DW$. Leg selectivities: $DP = 0.02$, $DE = 0.1$, $DS = 0.5$, $DW = 0.5$, that of DC varies from 0.01 to 0.25.

We run several instances of this query, varying the predicate selectivity on Leg DC from 0.01 to 0.25 by changing the literal, keeping selectivities on DP , DE and DS fixed (0.02, 0.1, 0.5). Figure 11 plots the elapsed time of No Merging, Eager Merging, and Lazy Merging as we vary the selectivity of Leg DC . Each bar shows the overall elapsed time, plus a break down into merge and intersect phases (indicated by shading). The merge phase is any segment formation and merging that happens before intersection begins.

The first observation is that No Merging and Lazy Merging beat Eager Merging handily at low selectivities, and the benefit drops as we increase the selectivity. To understand why, we shown in Table 1 the number of leaf page IOs that No Merging and Lazy Merging save over Eager Merging. As the selectivity increases, the number of leaf page IOs for Lazy Merging (and No Merging) increases much more sharply than for Eager Merging.

This behavior follows from the way Lazy merging uses high-rids to avoid segment formation. With a uniform distribution, the RIDs of Orders table are scattered uniformly across all the segments. But at low selectivities the combined number of RIDs that used to lookup() into each leg is still small, and will find no match in many leaf pages. Lazy merging avoids IO on these pages using the high-rids information in the non-leaf pages.

Due to this IO saving and the associated reduction in parsing cost, Lazy Merging and Eager Merging win at low selectivities. No Merging starts to lose to Eager Merging at selectivities beyond 0.20, due to the increased number of lookups in unmerged segments. However, Lazy Merging remains competitive with Eager Merging even at higher selectivities. Since all the dimension legs are uniform distributed, all segments within a leg are roughly the same size, and Lazy Merging does not help to preferentially merge some smaller segments. But still, segment sizes vary *across legs*, and Lazy Merging results in the legs with a small average segment size being merged early (almost like Eager Merging), while the legs with a large average segment size are merged very late (almost like No Merging).

Notice that the cost of Eager Merging is mainly in the merge phase – because it merges segments up front, while that of Lazy Merging and No Merging arise primarily from the intersect phase. This highlights a secondary advantage of Lazy Merging, that it *outputs results in a pipelined fashion*, in contrast to Eager Merging. Both Lazy Merging and No Merging algorithm incur a small up front merge cost. This is for the few segments that are contained entirely in one page. Since the high-rids information is not helpful for these segments, they have to be scanned anyway. We scan and merge such segments in the beginning, before intersection begins.

4.1.2 Uniform Distribution With Bad Residual

We now consider a slight variation of the previous experiment that highlights the utility of Lazy Merging when the optimizer makes a bad decision. We run a query where the optimizer adds to the index anding phase a dimension that is both poorly selective, and has very few segments. Ideally, such a dimension should be joined as a residual after the index anding phase.

We repeat the previous experiment by adding an additional table DW with the following predicate $DW.city = \langle value \rangle$ to the query, and forcing a query plan where leg DW is intersected during index anding. Figure 12 plots the elapsed time of Eager, Lazy, and No Merging. Observe that the elapsed time for Eager Merging has been doubled or even tripled with the addition of DW . On the other hand, the elapsed time for No Merging and Lazy Merging have only increased modestly. In particular, notice that No Merging beats Eager Merging even at high selectivities (0.25 on DC), in contrast to the previous experiment (Figure 11). The poor performance of Eager Merging arises from the large cost of merging the the segments of DW : DW has 5M rids even after its local predicate has been applied. On the other hand, No Merging and Lazy Merging perform well because the intersection of DP , DC , DE and DS has already produced a very small list of matching RIDs, and these can be used to skip over most of the pages of DW .

Thus, Lazy Merging acts as a safeguard against bad residuals and improves the query predictability.

4.1.3 Uniform Distribution With Rid Correlation

We now evaluate how addition of a predicate correlated with rid affects the merging algorithms.

Dimension table DT in our schema is a time dimension, and its values are naturally correlated with RID (newly inserted records tend to have higher dates). We add DT to our query (we remove

Section	selectivity	No Merging				Eager Merging				Lazy Merging			
		Elapsed time		# of IOs		Elapsed time		# of IOs		Elapsed time		# of IOs	
		Merge	And	Merge	And	Merge	And	Merge	And	Merge	And	Merge	And
4.1.1	0.02x0.1x0.5x0.01	0.125	0.685	1950	2998	1.843	0.133	11032	0	0.125	0.225	1950	2998
	0.02x0.1x0.5x0.05	0.125	1.589	1950	5303	2.251	0.347	12324	0	0.125	0.793	1950	5303
	0.02x0.1x0.5x0.10	0.125	2.152	1950	7441	2.653	0.381	13613	0	0.125	1.117	1950	7441
	0.02x0.1x0.5x0.15	0.125	3.126	1950	9430	3.040	0.396	14901	0	0.125	1.381	1950	9430
	0.02x0.1x0.5x0.20	0.125	4.268	1950	11664	3.501	0.425	16224	0	0.125	1.883	1950	11664
	0.02x0.1x0.5x0.25	0.125	5.906	1950	13472	3.914	0.434	17515	0	0.125	2.183	1950	13472
4.1.2	0.02x0.1x0.5x0.01x0.5	0.140	1.034	1950	7676	6.312	0.157	23942	0	0.140	0.491	1950	7676
	0.02x0.1x0.5x0.05x0.5	0.140	2.459	1950	17319	6.68	0.368	25234	0	0.140	1.542	1950	17319
	0.02x0.1x0.5x0.10x0.5	0.140	3.302	1950	19594	7.044	0.418	26519	0	0.140	1.908	1950	19594
	0.02x0.1x0.5x0.15x0.5	0.140	4.289	1950	21592	7.458	0.448	27810	0	0.140	2.237	1950	21592
	0.02x0.1x0.5x0.20x0.5	0.140	5.117	1950	23561	7.936	0.462	29102	0	0.140	2.560	1950	23561
	0.02x0.1x0.5x0.25x0.5	0.140	6.100	1950	25361	8.277	0.476	30392	0	0.140	2.832	1950	25361
4.1.3	0.02x0.1x0.5x0.01x0.1	0.140	0.115	1950	1560	1.940	0.017	12374	0	0.140	0.088	1950	1781
	0.02x0.1x0.5x0.05x0.1	0.140	0.219	1950	2081	2.337	0.040	13668	0	0.140	0.167	1950	2304
	0.02x0.1x0.5x0.10x0.1	0.140	0.289	1950	2459	2.755	0.047	14957	0	0.140	0.239	1950	2683
	0.02x0.1x0.5x0.15x0.1	0.140	0.386	1950	2823	3.126	0.051	16245	0	0.140	0.310	1950	3047
	0.02x0.1x0.5x0.2x0.1	0.140	0.451	1950	3179	3.598	0.054	17532	0	0.140	0.380	1950	3403
	0.02x0.1x0.5x0.25x0.1	0.140	0.581	1950	3505	3.908	0.057	18823	0	0.140	0.466	1950	3729

Table 1: Elapsed Time and IOs for experiments of Section 4.1.1, 4.1.2, 4.1.3, split into merge and intersect (“and”) phases

DW, so the query is otherwise like in Section 4.1.3), with a predicate with selectivity 0.1, and repeat the same measurements as the previous two experiments. Figure 13 compares the elapsed time of Eager Merging, No Merging, and Lazy Merging with the addition of leg DT. Notice that No Merging and Lazy Merging gain huge benefits from rid correlation, while the elapsed time of Eager Merging improves by little. The main disadvantage for No Merging is the increased cost of separate lookup(s) in each unmerged segment. However, when rid correlation happens, the unmerged segments have very little overlapping rid-range with one another. With the aid of High-RID, the cost of separate lookup(s) are kept cheap in most unmerged segments. On the other hand, Eager Merging has to pay the expensive cost of merging the segments.

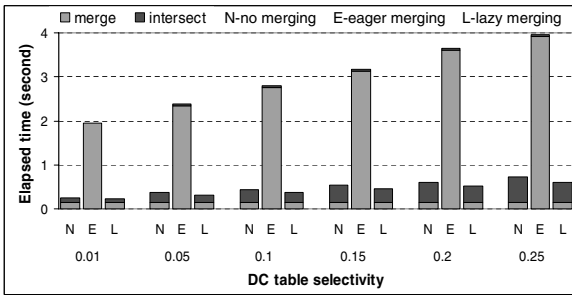


Figure 13: Lazy, Eager, and No Merging for $DC \cap DP \cap DE \cap DS \cap DT$. DT is correlated with RID . Leg selectivities: $DP = 0.02$, $DE = 0.1$, $DS = 0.5$, $DT = 0.1$, $DC = 0.01$ to 0.25 .

4.1.4 Skewed Distribution With No Rid Correlation

To evaluate the impact of skew, we add one more dimension leg DA (Account) into the query, with the Zipf distribution on $Orders.account_id$. The other four original legs (DP, DC, DE, and DS) still maintain uniform distributions. We re-run the same experiments by varying the selectivity on DC.

Figure 14 shows that similar to rid correlation, skew also gives advantages to Lazy Merging. Due to skew, the segments of leg DA

have varying sizes, and large-sized segments present an opportunity for Lazy Merging to avoid scanning pages. However, skew is not so helpful for No Merging. This is because skew also creates many small segments. For No Merging, these small segments introduce extra lookup(s), which offset the benefits from scan avoidance in the large segments. Whereas, Lazy Merging merges these small segments early and thus performs better.

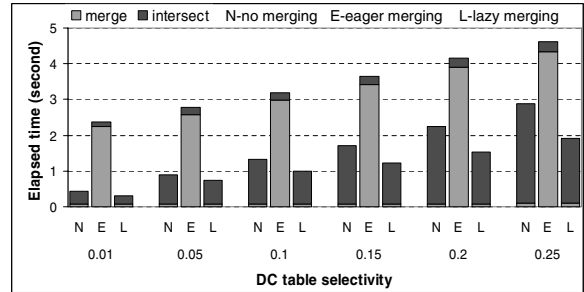


Figure 14: Effectiveness of Lazy Merging on Skewed Data for $DC \cap DP \cap DE \cap DS \cap DA$. DA has skewed data. Leg selectivities on $DP = 0.02$, $DE = 0.1$, $DS = 0.5$, $DA = 0.1$, and that of DC varies from 0.01 to 0.25 .

4.2 Predictability from Adaptive Intersection

Section 4.1 examined the value of lazy merging, and we now turn to the effects of adaptive intersection. The traditional best-AND-tree approach relies on the optimizer having the right statistics to pick the best AND-tree. However, the optimizer may often choose a poor AND-tree due to wrong statistics. In this section, we evaluate the robustness of intersection algorithms when the optimizer chooses a poor AND-tree. To focus on the intersection performance alone, we use Eager Merging in all the experiments in this section. The numbers we report are the intersection times: the merge phase is the same for all intersection algorithms.

We use 4 dimension tables from the previous section: DP, DC, DE, DS, and add 6 extra dimensions. The first four have uni-

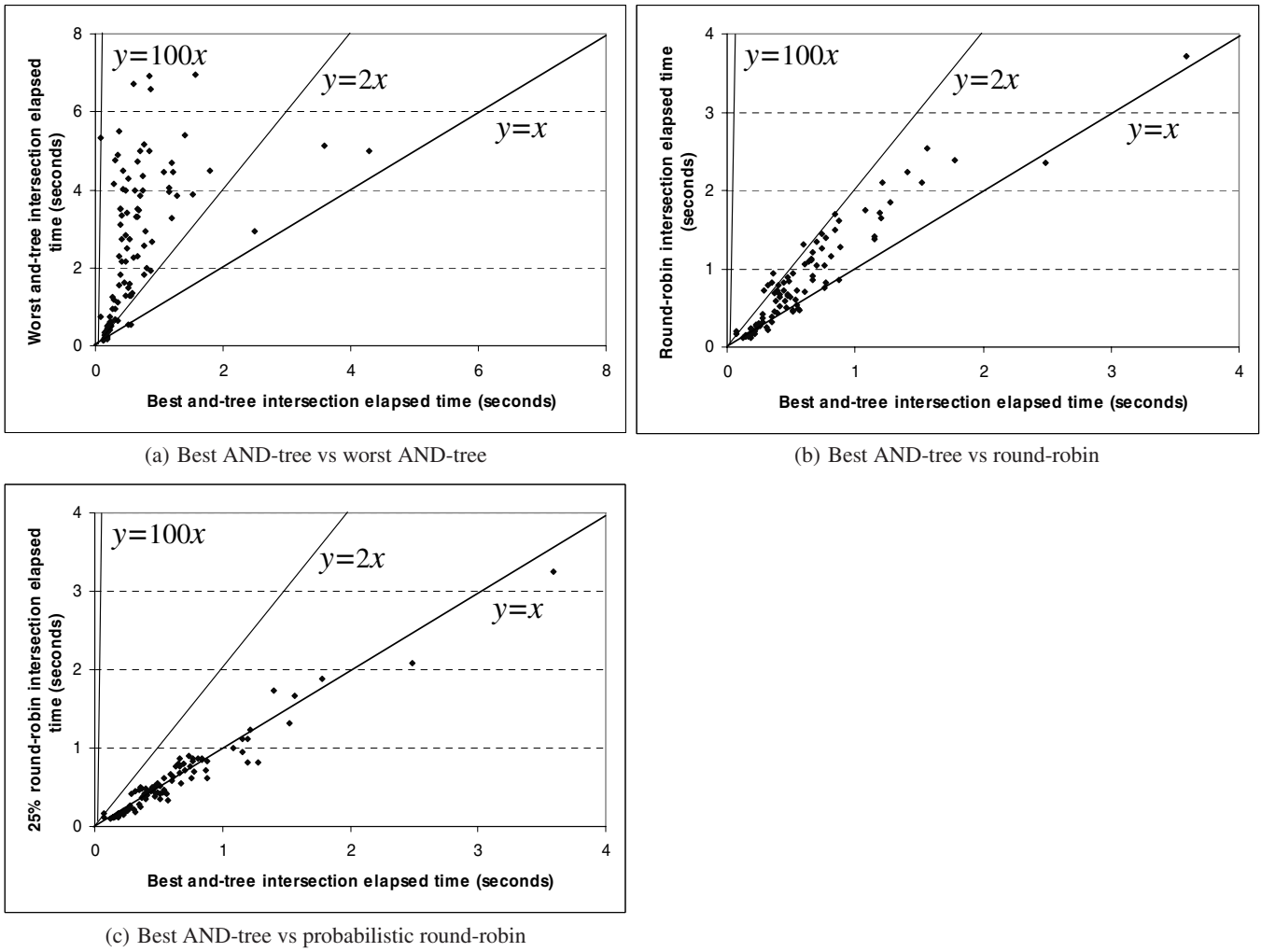


Figure 15: Comparison of 4 intersection algorithms on 100 randomly generated queries

Name	Cardinality	Distribution of foreign keys
FactTable (O)	10,000,000	-
Product (DP)	100,000	Uniform
Customer (DC)	10,000	Uniform
Employee (DE)	500	Uniform
Store (DS)	10	Uniform
DFW1	1000	Forward RID Correlation
DFW2	100	Forward RID Correlation
DFW3	10	Forward RID Correlation
DBW1	1,000	Backward RID Correlation
DBW2	100	Backward RID Correlation
DBW3	10	Backward RID Correlation

Table 2: Tables Used in Adaptive Intersection Experiments

form distribution (in the corresponding foreign key column in Orders). The six new dimensions have correlation with RID. Three of these have forward RID correlation (meaning that the predicates on these legs selects foreign keys that occur more frequently in higher RIDs), and three have backward RID correlation (the converse). Table 2 shows the cardinality and skew of each leg.

Since our goal is to study the predictability of query performance, we generate 100 queries by randomly choosing the number

of dimensions to join, which dimensions to join, and their selectivities. For each query, we produce the best AND-tree and the worst AND-tree query plan based on the selectivities.

Then we run the 100 queries under four intersection scenarios, 1) best AND-tree intersection, 2) worst AND-tree intersection, 3) round-robin intersection using the worst AND-tree, and 4) probabilistic round-robin using the worst AND-tree. We use the first scenario as the base, and compare results from the last three to the best AND-tree intersection. Our goal is to find how bad it can be for the optimizer to pick the wrong AND-tree, and how well round-robin and probabilistic round-robin intersection do in this case.

Figure 15 (a) compares the performance of the best AND-tree and the worst AND-tree for each query. Each dot represents a query, and its X-coordinate is the elapsed time for the best AND-tree plan while the Y-coordinate is the elapsed time for the worst AND-tree plan. We also plot three slopes $y = 100x$, $y = 2x$, $y = x$ for reference. Notice that for some queries, the elapsed time for the worst AND-tree plan is 100 times the elapsed time for the best AND-tree plan. That means these queries are very sensitive to the order of the AND-tree; if the optimizer chooses the wrong order, the queries will be up to 100 times slower.

Figure 15 (b) is the same plot, with worst AND-tree replaced by round-robin intersection. The points are now clustered almost

exclusively between the $y = 2x$ and $y = x$ lines, showing that the performance is always within a factor of two of the best possible.

Figure 15 (c) demonstrates the results for our new probabilistic intersection, which does round-robin part of the time, and lookup(s) in the best leg otherwise. Most queries are contained near or below $y = x$ line, i.e., the performance is as good the best elapsed time using an AND-tree. This highlights the predictability of this intersection algorithm

For more than a half of the 100 queries, the learning algorithm beats the elapsed time of the best static AND-tree. This is because the selectivity on the dimensions is different in different rid ranges. As we do lookup(s) in intersection, the learning algorithm dynamically finds the most two selective legs and makes the largest jump, while the static AND-tree has to stick to a fixed order of legs and the round-robin has to go around all the legs each time.

4.3 Combined performance result

Finally, we present the overall performance of our proposed index anding method: Lazy Merging plus adaptive intersection with probabilistic round-robin.

We repeat the experiment of 100 randomly generated queries derived from the 10 index legs. Figure 16 shows the overall elapsed time, including both merging and intersection phases. The X-coordinate is the elapsed time using intersection with the best static AND-tree, and Eager Merging. The Y-coordinate is the elapsed time using the new adaptive intersection and Lazy Merging algorithm.

Notice that the majority of the points are well below the $y = x$ line. For these queries, lazy adaptive intersection beats the traditional eager AND-tree intersection, reducing the overall elapsed time by a factor of 5 in many cases. There are some points above the $y = x$ line, indicating performance regression. But all the 100 queries fall under the $y = 1.8x$ line, which means, our proposed lazy merging and adaptive intersection algorithm always performs within 1.8 times of what is achieved by the best AND-tree, even though we assume *no knowledge of cardinalities*.

We have done some further analysis of this regression. Our findings are that the queries with regression are queries over very few dimensions. When a query does not have many dimension legs, our algorithms have few opportunities to enjoy the benefits from Lazy Merging or adaptive intersection.

So we repeat the overall elapsed time experiments on 50 new queries that involve all 10 legs. We generate these 50 queries by randomly choosing the selectivities on each leg. The results are shown as Figure 17. Using the traditional Eager Merging and static AND-tree approach, the 50 queries all take 15 to 17 seconds. However, with our adaptive intersection and Lazy Merging algorithm, the overall elapsed time shrinks to the range between 13 and 15 seconds. That shows our approach is more likely to beat the traditional method when the queries involves more legs.

5. RELATED WORK

Index anding has been used in query processors for a long time. One early paper [9] proposed the use of an AND-tree for accessing a single table using multiple indexes. While the basic algorithm accesses indexes in a sequential manner, the paper also exploits parallelism by using one process for each index access. This parallel approach supports dynamic adaptation by choosing the intersection legs at the run-time and turning off an unselective leg. The sequential approach relies on the optimizers good estimate of statistics. The parallel alternative avoids the dependency on base table statistics, but implicitly makes an independence assumption about the size of the intermediate intersections. It does not address changing leg selectivities either.

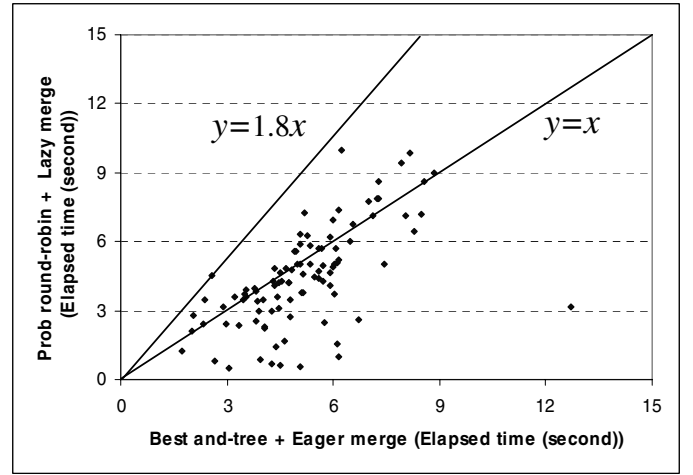


Figure 16: Performance of Lazy Merging with Adaptive Intersection on queries over random subset of dimensions

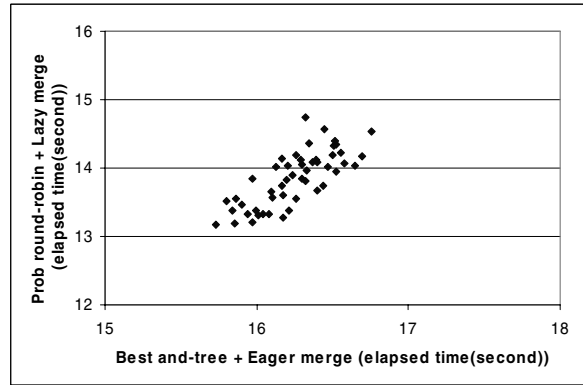


Figure 17: Performance of Lazy Merging with Adaptive Intersection on a query over all 10 dimensions, with randomly chosen selectivities

A popular way to do index anding is with bitmapped indexes (e.g., [11]). These are widely used in data warehouses and alleviate some of the problems of index anding due to their high efficiency. But they are not suitable for evaluating some predicates, and have a high maintenance cost when the update volume is high. Our lazy merging algorithm can improve the performance of index anding using B-Tree indexes, and our adaptive intersection algorithm can work well with both bitmap and B-Tree indexes.

In XML databases, RID-list intersection is used in finding all the matching occurrences for a twig pattern that selection predicates are on multiple elements related by an XML tree structure. [5] proposed a holistic twig join algorithm, *TwigStack*, for matching an XML twig pattern. IBMs DB2 XML has implemented a similar algorithm for its XANDOR operator [2]. *TwigStack* is similar to round-robin intersection, navigating around the legs for results matching a pattern. Our probabilistic round-robin algorithm can be applied to this context also. Lazy Merging is not as relevant in the XML context since the number of OR segments is usually small, and No Merging works fine.

A completely different way of doing star joins is to use multi-column indexes. If a single index meets most of the predicates, the remaining predicates can be treated as residuals and there is no

need for intersection at all. This method is very efficient with the right indexes, but needs a DBA to pick the “right” multi-column indexes and clustering for the workload, and often causes a high administrative burden. A large number of multi-column indexes also increases the update maintenance overhead. In contrast, index anding relies on single-column indexes. The number of possible single-column indexes is small enough that are built as a default on foreign key columns.

The merge phase of sort-merge joins is usually done by a linear search in the join inputs. Kamath and Ramamritham [8] present a bucket skip merge join that divides the sorted inputs into buckets that contain monotonically increasing values and maintain the value range for each bucket. The merge phase then uses the value range to skip whole or partial buckets, similar to the way we use the high-rid information in the B-Tree. This approach is similar to No-Merging algorithm; however, it does not address the tradeoff between sort merge cost and RID lookup cost in unmerged segments. In comparison, our Lazy-Merging algorithm considers this tradeoff by dynamically determining if we should merge segments.

There has been some recent theoretical work on adaptive set intersection, motivated by the problem of AND queries in search engines. Demaine et al [6] introduced and analyzed a round robin set intersection algorithm that we use as the basis for our leg intersection. Subsequently, Barbay et al [3] have generalized the analysis and also done further experimental studies on Google logs and TREC data [4]. Our probabilistic intersection is a variant on this algorithm. Merging is less important in this problem because ORs in search engine queries usually have only a small number of terms. But the intersection problem is the same, and we believe there are many opportunities for cross-pollination between this literature on set intersections and that on index anding.

Another area where adaptive ordering crops up is in selection join ordering. [10] present an adaptive algorithm for ordering pipelined selections even when there are correlations. The main difference between this problem and the index anding problem is that in selection ordering the input table is coming in as a stream and flowing through the selection operators, so each selection operator is like a leg that only supports contains() (as opposed to lookup()). Still, adaptive selection ordering is also useful in index anding, for deciding on the order among the residual tables. There are also some proposed mechanisms for dynamically adapting join orders (like [1]), but no well analyzed adaptation policies yet.

6. CONCLUSION AND FUTURE WORK

RID-list intersection is a widely used operation in query processing, from star joins to column stores.

We have presented two new algorithms for two phases of this operation: a lazy merging algorithm for forming sorted RID-lists of rows matching each predicate, and an adaptive intersection algorithm for intersecting these RID-lists. Deferred merging is considerably more IO efficient than the usual approach of eager merging. But it can be CPU-inefficient due to increased number of lookups into unmerged segments, so we proposed a lazy merging algorithm that monitors the cost of these extra lookups and merges segments when this cost gets too high. The adaptive intersection does not rely on selectivity estimates to choose an AND-tree. Rather, it dynamically learns which dimensions are more selective as the intersection proceeds and preferentially uses these dimensions to sweep the space of RIDs.

Experimentally, our methods perform much better than traditional eager merging and AND-trees, with the improvement being higher for skewed or correlated data. Our methods are also more predictable, in that they never perform much worse than what the best AND-tree would.

Looking forward, we think that predictability and competitive analyses should be a general theme of query processing research because of the difficulty of estimating cardinalities correctly. Both components of our proposed method have bounded competitive ratios: lazy merging is always within a factor of 2 of the best possible (in hindsight) merging policy, and round-robin intersection is within a factor k of the best possible static AND-tree when there are k dimensions. We plan to apply such round-robin techniques to the general problem of $n - ary$ joins, with a target of achieving competitive bounds.

Acknowledgement: We thank Gene Fuh for introducing and explaining this problem to us. Jeff Berger, David Chambliss, Gene Fuh, Prashant Pandey, Thomas Phan, and Garret Swart provided significant assistance in designing and implementing our algorithm. Finally, we thank the anonymous referees for several useful suggestions to improve the paper presentation.

7. REFERENCES

- [1] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.
- [2] A. Balmin, T. Eliasz, J. Hornibrook, L. Lim, G. M. Lohman, D. Simmen, M. Wang, and C. Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2), 2006.
- [3] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, 2002.
- [4] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Intl. Workshop on Experimental Algorithms*, 2006.
- [5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, 2002.
- [6] E. Demaine, A. López-Ortiz, and J. Munro. Adaptive set intersections, unions, and differences. In *SODA*, 2000.
- [7] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.
- [8] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report UM-CS-1996-020, University of Massachusetts at Amherst, 1996.
- [9] C. Mohan, Don Haderle, Yun Wang, and Josephine Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings of the International Conference on Extending Database Technology*, 1990.
- [10] Shivnath Babu, Rajeev Motwani, Kamesh Mungala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.
- [11] P. O’Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, 1997.
- [12] Oracle White Paper. Query Optimization in Oracle Database10g Release 2, June 2005.
- [13] SAP. SAP Business Information Warehouse Benchmark.
- [14] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO: DB2’s LEarning Optimizer. In *VLDB*, 2001.
- [15] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Emerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.