Data Management

Maurizio Lenzerini

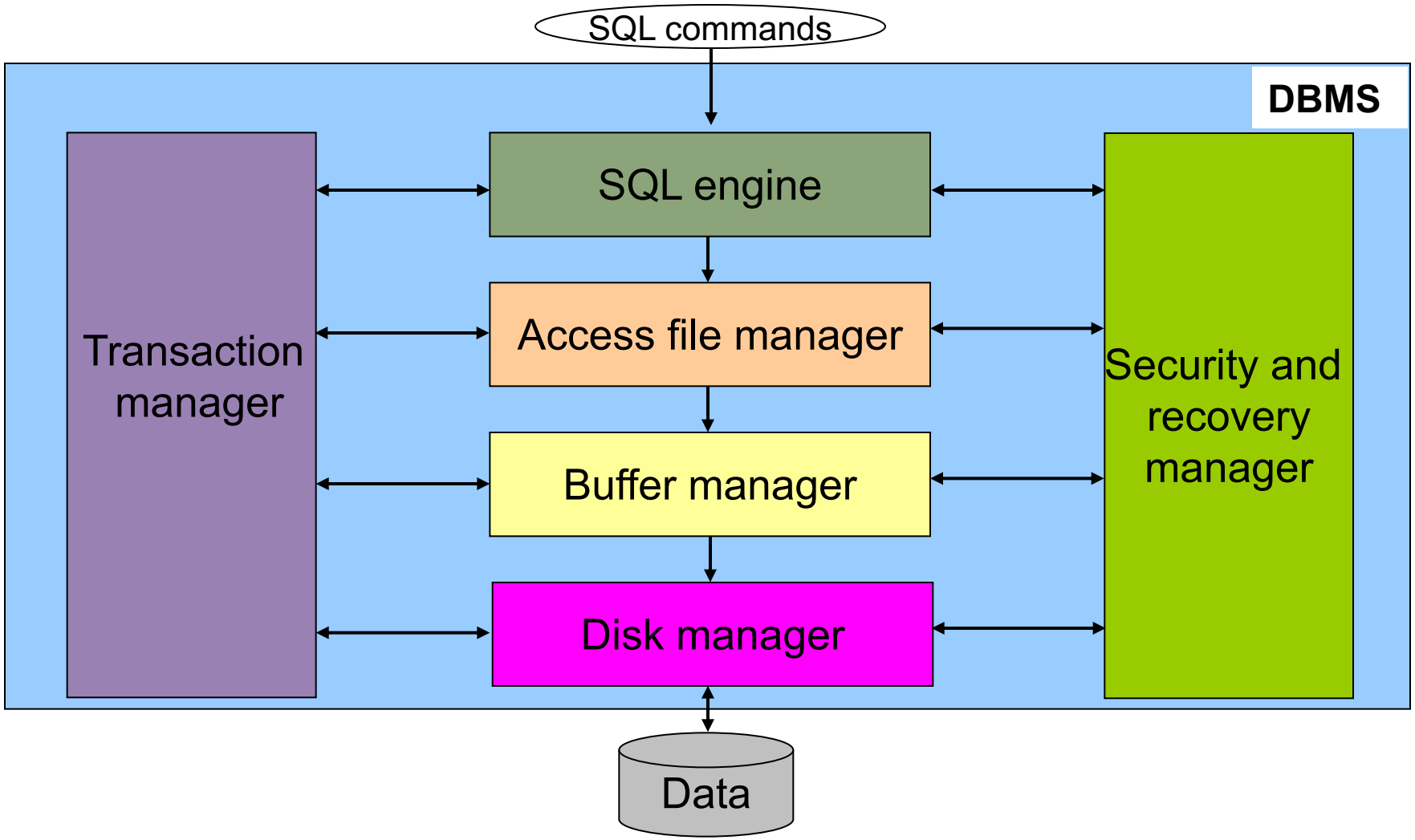
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

Academic Year 2018/2019

Part 6
Query processing – evaluation of operators

http://www.dis.uniroma1.it/~lenzerini/index.html/?q=node/53

Architecture of a DBMS



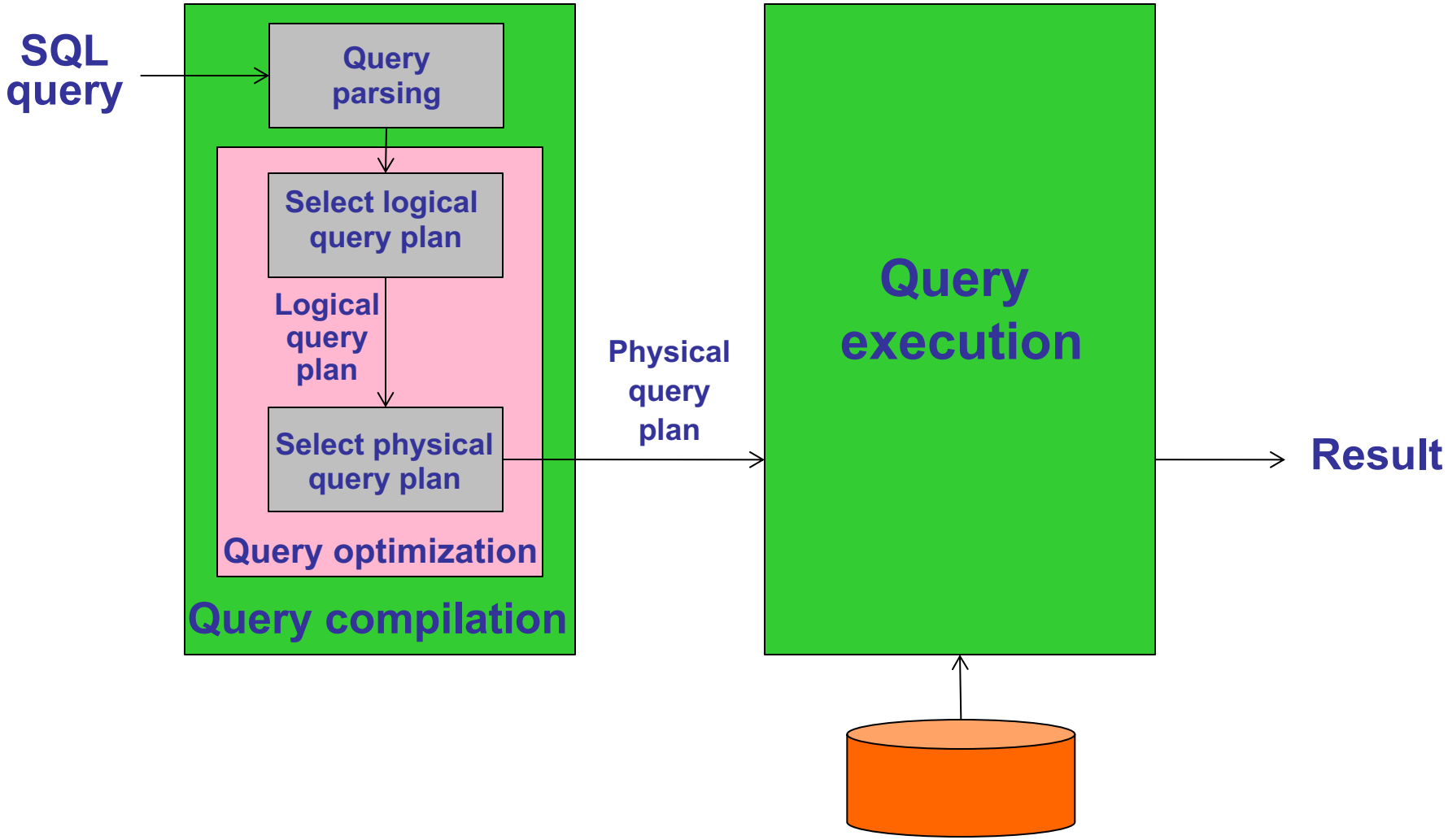
4. Query processing – evaluation of operators

4.1 Overview of query processing
4.2 Evaluation of relational operators

4. Query processing – evaluation of operators

4.1 Overview of query processing
4.2 Evaluation of relational operators

Query processing by the SQL engine



4. Query processing – evaluation of queries

4.1 Overview of query processing
4.2 Evaluation of relational operators

Query execution

Our goal is to understand how an SQL engine evaluates a query on the basis of the physical query plan it has decided.

The physical query plan indicates both the order of evaluation of the various subparts of the query, and which method the system should use for evaluating the various operators present in the query plan.

Therefore, we now study the second aspect, i.e., which are the algorithms that we can use for evaluating relational algebra expressions constituted by with just one operator. The operators we will consider are those of the relational algebra, plus the other operators deriving from the formulation of the query in terms of SQL.

Evaluation of relational operators

- Union, intersection and difference

  on both sets and bags (where bags are like sets, but with duplicates); we use the subscript S to indicate an operator that works on sets and produces a set. In the case where it works on sets or bags and produce bags, we use the subscript B
- Selection and projection
- Cartesian product and join (with all the variants of join)
- Duplicate elimination (turning a bag into a set)
- Grouping

  corresponding to GROUP-BY and the aggregation operators

  (SUM, AVERAGE, etc.)
- Sorting

  corresponding to ORDER-BY

Algorithms for operators

- We are intested in characterizing the different algorithms that can be used to implement the above mentioned operators.
- Indeed, in a physical plan, whenever an operator has to be applied to the corresponding operand(s), there is an explicit specification of which is the algorithm that has to be used to implement the operation.
- Examples of algorithms (see later for a detailed study):
  - table-scan
  - sort-scan
  - index scan
  - etc.

Materialization or pipeline

- In principle an operator can be applied in two different modes:
  - materialization-based
  - pipeline-based (or, on-demand, or tuple-by-tuple)

- The materialization mode is the obvious one: the system computes the result and stores it in a temporary relation

- The pipeline-based mode is an implementation of the operator that uses the notion of iterator, which is a mechanism for computing the result of an application of an operator in a tuple-by-tuple, on demand fashion. Every time an iterator is called, it is supposed to return (if possible) a tuple in the result of the application of the operator.

Iterator

- An iterator associated to an operator is an abstraction of objects that are manipulated by three functions:
  - Open – this function initializes all data structures (if any) used for getting the tuples that are the results of application of the operator, and prepares the process of getting such tuples (without actually getting any)
  - GetNext – this function returns the next tuple of the result, if any, and adjust the data structures so as to get subsequent tuples. It uses a boolean variable Found that is true if and only if a new tuple has been returned
  - Close – this function ends the iteration after all tuples have been obtained

Example: ScanIterator (for table-scan)

```
Open(R) { self.Rel := R; self.b := first block of Rel;     -- we assume Rel non-empty
          self.t := the first tuple of block self.b;
          FOUND := true;
         }
GetNext() { IF (self.t is past the last tuple on block self.b of self.Rel) {
                  IF (no next block in Rel) { FOUND := FALSE; RETURN; }
                  ELSE { increment self.b to the next block of self.Rel;
                           self.t := first tuple of block self.b of self.R;
             }
             self.oldt := self.t;
             let self.t be the next tuple of b;
             RETURN self.oldt;
           }
Close() { }
```

Example: BagUnionIterator (for bag-union)

Note: The iterator for bag union uses the iterator for table-scan shown in the previous slide.

```
Open(R,S) { self.R1 := R; self.R2 := S; ScanIterator c1 := Open(R1);
            ScanIterator c2 := Open(R2); self.curRel := R1; }

GetNext() { IF (self.curRel = R1) {
                self.t := c1.GetNext();
                IF (FOUND) RETURN self.t;
                ELSE {ScanIterator c2 := Open(R2); self.CurRel := R2; }
              }
            RETURN c2.GetNext(); // note that if self.R2 is exausted, FOUND is
                                 set to FALSE by c2.GetNext();
          }

Close() { c1.Close(); c2.Close(); }
```

Classification of algorithms for operators

The classification we use here refers to the number of passes of reading from secondary storage.

- Sequential methods
  - One-pass

    these algorithms read the data only once from secondary storage: usually (but not always), they work only when at least one of the operands fits in main memory (i.e., the buffer).

  - Nested-loop algorithms

    these algorithms are based on a loop (to analyze one relation) inside of which there is another loop (for analyzing the same relation or another one, depending on the operation). When specialized for the join operator, they can be seen as “one-pass and a half” algorithms, in the sense that they work when only one of the two operands fits in available main memory.

Classification of algorithms for operators

- Two-pass

  these algorithms read data a first time from secondary storage, process them in some way, then write intermediate data on secondary storage, read them again for further processing. They work for data that is too large to fit in available main memory.

- Index-based

  these algorithms base their strategy on the use of one or more indexes.

- Multipass

  these algorithms use three or more passes on the data and are natural generalizations of the two-pass algorithms

- Parallel method

  Parallel algorithms use many processors in a shared-nothing architecture

Classification of algorithms for operators

With regard to the technique used, three categories of algorithms will be particularly important:

- Sorting-based

  these algorithms requires to sort one or both operands. Some of two-pass or multipass algorithms use this technique.
- Hash-based

  these algorithms requires the use of one or more hash functions. Some of two-pass or multipass algorithms use this technique.
- Index-based

  these algorithms requires the use of one or more indexes. Obiously, all index-based algorithms satisfy this definition. However, there are two-pass or multipass algorithms that use one or more indexes, and therefore fall also in this category.

A note on the number of pages for a relation

In what follows, we denote by B(R) (or simply B, when R is understood) the number of pages used to store the relation R in secondary storage.

It is interesting to compare B(R) with the number QB(R) of pages required to store the records of R in secondary storage. Suppose that the average size of space per page occupied by each record of R is SR(R), the size of a page is SP, and the number of records for R is NR(R). Then we have that QB(R) = NR(R) × SR(R) / SP.

Since the pages containing the records of a relation R can be organized in different ways, we now analyze the different cases and see how B(R) is related to QB(R):

1. When the relation R occupies pages with only its records, and there is no free space specifically foreseen in such pages, then B(R) is the minumum number of pages holding its tuples, i.e., B(R) = QB(R). Note that in this case we say that the relation is clustered (not to be confused with the notion of clustered file or clustering index).

A note on the number of pages for a relation

2. When the pages occupied by relation R have only records of R, and there is free space specifically foreseen for them, i.e., the C% of the space in each page is free space, then B(R) = QB(R) × 100 /(100-C).

3. When the relation R is represented by a clustered file, and R is the "child" relation, i.e., in every page we have an average of C% of space not occupied by R (i.e., occupied by the "father" relation S, or free), then B(R) = QB(R) × 100 /(100-C) again.

4. When the relation is represented by a clustered file, and R is the "father" relation, i.e., in every page we have an average of C% of space occupied by such father, then B(R) = QB(R) × 100 / C.

In what follows, if not otherwise stated, we assume that the relations are clustered.

A note on the notions of “cluster”

We have seen different interpretations of the term “cluster” in the course. Let us recall the various interpretations:

1. Clustered-file organization

   A method by which the records of two relations co-habit in the same page. In particular, in the pages of a clustered-file organiztion, one record of the “father” relation is followed by a set of related records of the “child” relation

2. Clustered relation

   A relation that is stored in a set of pages that are exclusively, or at least predominantly, devoted to store the record of that relation.

3. Strongly clustering index (our main interpretation)

   An index on a relation R whose values are sorted coherently with the order used to store the records of the relation R

4. Weakly clustering index

   An index I is weakly clustering if all the tuples of the indexed data file with a fixed value for the search key used in I appear on roughly as few pages as can hold them. Note that our strongly clustering implies weakly clustering

The sequential algorithms that we will analyze

In the following, we will study the following classes of sequential algorithms:

- One-pass
  - for non-sorted relations
  - for sorted relations
- Nested-loop
- Two-pass
  - based on sorting
  - based on hashing
- Index-based
- Multi pass

In the cost analysis in terms of page accesses, we ignore the cost of page accesses due to the need of writing the result in secondary storage. Also, we mostly assume to work in the materialization-based mode, and therefore we consider the use of the output buffer frame, when needed. Note that if we carry out the analysis assuming the tuple-by-tuple mode, then the output buffer frame is not always needed.

One-pass algorithms

- Projection on relation R

  We read the pages of R one at a time into a buffer frame (input frame), perform the operation on each tuple, and write the projected tuples into another buffer frame (output frame). Whenever the output frame is full, we write it into the result in secondary storage.

  Space in buffer: one input frame and one output frame.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B (i.e., table-scan is used for analyzing the tuples of R) .

- Selection on relation R (R not sorted and with no index)

  Similar to projection: we read the pages of R one at a time into a buffer frame (input frame), perform the operation on each tuple, and write the selected tuples into another buffer frame (output frame). Whenever the output frame is full, we write it into the result in secondary storage.

  Space in buffer: one input frame and one output frame.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B (i.e., table-scan is used for analyzing the tuples of R).

One-pass algorithms

- Selection on relation R (R sorted on the attributes used to search and with no index)

  This is the case where the "where" condition refers to the search key on the basis of which R is sorted (otherwise sorting is irrelevant). We use binary search or interpolation search.

  Space in buffer: one frame.

  Cost of I/O: if the size of the relation R is B pages, then the cost of finding the first record is $O(\log_2 B)$, in the case of binary search. If the search key is not a key of the relation, we have to add as many page accesses as the number of pages of interest.

One-pass algorithms

- Duplicate elimination on R (R not sorted and with no index)

  We can read each page of R one at a time, and for each tuple we have to check whether we have already seen it. For this purpose, we need to keep in the buffer one copy of every tuple we have seen. The data structure T that we can use is either a hash table or a balanced binary tree (if we use a list, then the complete operation would take $O(N^2)$ tuple comparisons, where N is the number of tuples). At the end, we write into the result all the pages used for the data structure T.

  Space in buffer: if the available buffer frames are M, and we use one frame to read R, we can use this algorithm if the number of pages needed to store the non-duplicate tuples of R is not greater than M-1; globally, the space used will be M.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B.

One-pass algorithms

- Duplicate elimination on R (R sorted on any set of attributes and with no index)

  We read each page of R one at a time, and we keep in the buffer (without duplicates) the group of tuples with the value of the sorting attributes equal to the last value seen. For each tuple in the page we read, we have to check whether the tuple is equal to one of the tuples kept in the buffer. If yes, we ignore the tuple. If not, we add the tuple to the group. When the tuple under analysis belongs to a new group, we write the pages containing the tuples of the previous group into the result (except one page, if such page is not full yet and is not the last page to write).

  Space in buffer: if the available buffer frames are M, and we use one frame to read R (input frame), we can use this algorithm if the pages needed to store the largest group of tuples with the same value of the sorting attributes fit in M-1 buffer frames; in the worst case, the space used will be M. If the sorting attributes includes all the attributes of the relation, then we only need one input frame and one additional frame.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B.

One-pass algorithms

- Grouping on R (R not sorted and with no index)

  We read R one page at a time using one input frame, and we create in the buffer one entry for each group, i.e., for each value of the grouping attributes. The entry for each group consists of the values of the grouping attributes and an accumulated value or values for each aggregation. The accumulated value depends on the aggregation function (min or max of the values seen so far in the group, or the count so far, or the SUM so far, or the SUM and the count for the AVG).

  When all the tuples of R have been read, we can produce the output, by writing into the result all the frames containing the tuples for the groups. Until the last tuple is seen, we cannot begin to produce the output. This implies that in the iterator, the entire grouping is done by the Open function before the first tuple can be retrieved by GetNext.

  Note that processing in the buffer can be done efficiently if we use appropriate data structures (hash table, or balanced binary tree) for the entries.

  Space in buffer: if the available buffer frames are M, then we can use this algorithm if the number of groups is such that the number of frames needed to store one entry for each group is not greater than M-1 (one frame is the input frame); in the worst case, the space used will be M.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B.

One-pass algorithms

- Grouping on R (R sorted on the grouping attributes and with no index)

  We need to consider only the accumulated values for one group (the current one) in the buffer. We can analyze the tuples of R by reading the pages of R one page at a time. When the tuples of one group is finished, we write the tuple corresponding to the current group in the output frame. Whenever the output frame is full, we write it into the result in secondary storage.

  Space in buffer: one input frame and one output frame.

  Cost of I/O: if the size of the relation R is B pages, then the cost is B.

- Bag union

  To compute the bag union of R and S, we simply read every page of R and write it into the result, and then read every page of S and write it into the result.

  Space in buffer: one frame.

  Cost of I/O: if the size of the relation R is B(R) pages, and the size of relation S is B(S), then the cost is B(R) + B(S).

One-pass algorithms

- Other binary operators (with operands R and S not sorted)

  We read the pages of the smaller of the operands R and S into the buffer, and build in the buffer a suitable data structure (typically, a hash table or a balanced binary tree) in which tuples can be found and inserted quickly.

  Space in buffer: if the available buffer frames are M, then we can use this algorithm if the minimum between B(R) and B(S) is not greater than M-2 (one frame is for the input and one frame is for the output); globally, the space used will be M (as before, B(R) is the size of R, and B(S) is the size of S)

  Cost of I/O: the cost is B(R) + B(S).

We now analyze the various binary operators. If not otherwise stated, we will be assuming that $B(S) \leq B(R)$.

One-pass algorithms

- Set union

  We read the pages of S into M-2 buffer frames, we copy them into the result, and build a data structure in the buffer with the tuples of S. We then read each page of R into the (M-1)-th buffer frame, one at a time. For each tuple t of R, we use the data structure to decide if t is not in S. If t is not in S, then we copy it to the output frame, otherwise we skip it, and delete it from the buffer. Whenever the output frame is full, we write it into the result.

- Set intersection

  We read the pages of S into M-2 buffer frames, and build a data structure in the buffer with the tuples of S. We then read each page of R into the (M-1)-th buffer frame, one at a time. For each tuple t of R, we use the data structure to decide if t is in S. If t is in S, then we copy it to the output frame, and delete it from the buffer, otherwise we skip it. Whenever the output frame is full, we write it into the result.

One-pass algorithms

- Set difference

  To compute $R -_S S$, we read the pages of S into M-2 buffer frames and build a data structure in the buffer with the tuples of S. We then read each page of R into the (M-1)-th buffer, and for each tuple t of R we check if t is in S. If t is in S, we skip the tuple and delete it from the buffer, otherwise we copy it to the output frame. Whenever the output frame is full, we write it into the result.

  To compute $S -_S R$ we read the pages of S into M-1 buffer frames and build a data structure in the buffer with the tuples of S. We then read each page of R into the M-th frame, one at a time, and for each tuple t of R, if t is in S, then we delete it from the buffer. At the end, we copy into the result the pages with all the tuples of S that are in the buffer.

One-pass algorithms

- Bag intersection

  We read the pages of S into M-2 buffer frames, and build a data structure in the buffer that associates to each tuple a count representing how many times it appears in S. We then read each page of R into the M-th frame, one at a time. For each tuple t of R, we use the data structure to decide if t is not in S. If t is not in S, then we ignore the tuple, otherwise, we copy it to the output frame, and we decrement the count associated to it in the buffer. When this count is 0, we delete the tuple from the buffer. Whenever the output frame is full, we write it into the result. Note that in this case, it is sufficient that the set of tuples of S without duplicates (and not the whole S) fits in M-2 frames.

- Bag difference

  To compute $S -_B R$, we read the tuples of S into M-1 buffer frames, and build a data structure in the buffer that associates to each tuple a count representing how many times it appears in S. We then read each page of R into the M-th frame, one at a time. For each tuple t of R, we use the data structure to decide if t is in S. If t is in S, we decrement the count associated to it in the buffer. When this count is 0, we delete the tuple from the buffer. At the end, we copy into the result all the tuples in the data structure (whose count is positive), and the number of times we copy it equals that count.

  To compute $R -_B S$, we read the tuples of S into M-2 buffer frames, and when we read a tuple t of R, if it is not in the buffer (i.e., its count in the buffer would be 0), then we copy it to the output frame. Otherwise, we simply decrement its count by 1 (and when the count is 0, we delete it from the buffer). Whenever the output frame is full, we write it into the result.

One-pass algorithms

- Cartesian product

  We read the pages of S into M-2 buffer frames. We then read each page of R using the (M-1)-th frame, and for each tuple t of R we concatenate t with each tuple of S, and write the resulting tuples in the M-th frame (output frame). Whenever the output frame is full, we write it into the result.

- Natural join

  In this and in other join algorithms, let us assume that R(X,Y) is joined with S(Y,Z), where Y represents all the attributes that R and S have in common. We continue to assume that S is the smaller relation. We compute the natural join as follows:

  1. We read the pages of S into M-2 buffer frames, and store their tuples into a data structure in the buffer, such as hash table, or balanced binary tree, with the attribute Y as search key.
  2. We then read each page of R using the (M-1)-th frame. For each tuple t of R, we find the tuples of S that agree with t on all attributes in Y, using the data structure. For each matching tuple of S, we form a tuple by joining it with t, and write the resulting tuples in the M-th frame (output frame). Whenever the output frame is full, we write it into the result.

Exercise 1

Describe a generalization of the natural join algorithm to capture the following operators:

- the equi-join,
- the theta-join.

Exercise 2

(a) Suppose that R and S are sorted (in particular, on the primary key, or on the whole attributes). Do the algorithms for

- (set or bag) union,
- (set or bag) difference,
- (set or bag) intersection,
- cartesian product,
- natural join

change with respect the formulation given above?

(b) Pick up 2 or 3 operators that we have analyzed, and write the iterator for it, thus coming up with the corresponding algorithm for tuple-by-tuple computation.

Summary of one-pass algorithms (relations not sorted)

- B is the number of pages of the single operand
- R and S are the two operands → B(R) and B(S) are the numbers of their pages
- We assume that the relations are not sorted
- We ignore the cost of writing the result in secondary storage

Operator	# buffer frames M required	Cost (Disk I/O)
selection, projection,	*2*	*B*
bag union	*1*	*B*
duplicate elimination, grouping	*B + 1*	*B*
union, intersection, difference, cartesian product, join	*min(B(R),B(S)) + 1 or min(B(R),B(S)) + 2*	*B(R) + B(S)*

Summary of one-pass algorithms (sorted relations)

- We assume the relations sorted appropriately (see exercise 2)

Operator	# buffer frames M required	Cost (Disk I/O)
selection on sort key	*2*	*$\log_2 B$*
selection not on sort key, projection	*2*	*B*
bag union	*1*	*B*
duplicate elimination	*2 (if all attributes form the search key), B+1 (worst case)*	*B*
grouping	*2*	*B*
union, intersection, difference	*3*	*B(R) + B(S)*
cartesian product, join	*min(B(R),B(S)) + 2*	*B(R) + B(S)*

Nested-loop algorithms

We now discuss the nested-loop operations, which are algorithms whose schema is based on a loop (to analyze one relation) inside of which there is another loop (for analyzing the same relation or another one, depending on the operation).

For selection and projection, it is not reasonable to use a nested loop algorithm. So, we start with duplicate elimination.

1. Duplicate elimination

We consider an order on the pages of R. We read R one page P at a time (from the first in the order, to the last) in one input buffer frame, and for each page, we read in another input buffer frame all subsequent pages of R, one page at a time. For each tuple t of P, we write t once (ignoring duplicates in the page P) in the output frame if there is no other subsequent pages of R with the tuple t, otherwise we ignore all copies of t in P. Whenever the output frame is full, we write it into the result.

Space in buffer: 3 frames.

Nested-loop and block nested-loop algorithms

It is easy to see that the above duplicate elimination algorithm requires B(B+1)/2 page accesses.

We leave as an exercise to describe the nested-loop algorithms for set union, set intersection, set difference, bag intersection and bag difference, under the assumption that we use only 3 buffer frames. It is easy to see that under such assumption such operators require B(S) + B(R) × B(S) page accesses, where S is the smaller relation.

Nested-loop and block nested-loop algorithms

Let us go back to the duplicate elimination operator. If we can use M buffer frames (where M is greater than 1) besides the one for reading the pages of the outer loop relation and the one for the output, i.e., if we can use more than 3 buffer frames in total, we can still follow the idea of the nested-loop algorithm, but do much better in terms of performance. Indeed, instead of loading the pages of the relation one at a time in the outer loop, we can load a goup of M pages at a time. More precisely, we perform the load of the relation for the outer loop in "blocks" constituted by M pages, and for each of these blocks we perform the load of the subsequent pages of the same relation in the inner loop. Therefore we spend $(M + 2M + ... + M \times \text{ceil}(B/M)) = M \times (1 + 2 + ... + \text{ceil}(B/M)) = M/2 \times \text{ceil}(B/M) \times (\text{ceil}(B/M) + 1)$ page accesses, approximated as

$$B \times (3/2 + B/M) + M$$

(obtained by approximating ceil(B/M) by B/M + 1).

Following this idea also for the other operators, we obtain a set of variants of the nested-loop algorithms, called block nested-loop algorithms. We leave as an exercise to describe the block nested-loop algorithms for the various operators. It is not difficult to see that for the operators with two operands, the cost can be approximated as $B(S) + B(R) \times \text{ceil}(B(S)/M)$ approximated as

$$B(S) + B(R) \times (1 + B(S)/M)$$

page accesses, where S is the smaller relation.

Nested-loop and block nested-loop algorithms

Since the cost of page accesses is high, in general these algorithms are not often used, or they are used in special situations, for example if the system dynamically realizes that the number of free buffer frames are less than originally estimated at the beginning of the execution of the algorithm.

Notice, however, that the nested-loop algorithm is sometimes explicitly chosen by the SQL engine for the join operator. We discuss the nested-loop join next.

Exercise 3

- Design nested-loop algorithms for grouping and aggregation, set union, set intersection, set difference, bag intersection and bag difference, in the case where we can only use 3 buffer frames. Evaluate the cost of the algorithms.

- Design block nested-loop algorithms for grouping and aggregation, set union, set intersection, set difference, bag intersection and bag difference, in the case where we can use M buffer frames (for M generic). Evaluate the cost of the algorithms.

Nested-loop join algorithms

We now discuss the nested-loop join algorithm, where one of the arguments is read only once, but the other argument is read repeatedly. In this sense, although some still talk about one-pass algorithm, technically this algorithm is a "one and a half" pass algorithm (in the sense that one relation is read once, and the other is read more than once).

In the following we use the following symbols

- B(R), as usual, for the number of pages of R, p_R for the (average) number of tuples per page of R
- B(S), as usual, for the number of pages of S, p_S for the (average) number of tuples per page of S

and, in the example, we consider the following values: B(R)=1000, B(S)=500, p_R=100, p_S=10. Also, we assume 10 ms per page access. Moreover, when evaluating the cost of the join, we ignore the cost of writing the result.

Tuple-based nested-loop join algorithm

We start with the simplest variant of the nested-loop join.

1. Tuple-based nested-loop join

   To compute the natural join of R(X,Y) and S(Y,Z)

   we do as follows (the outer relation is the smaller):

```
Scan the outer relation S, and
        for each page G of S and for each tuple E in G:
                Scan R, looking for the tuples "joining" E
```

The cost is $B(S) + (p_S \times B(S) \times B(R))$, which is very high. Indeed, the cost of the nested loop join in our example is $500 + (100 \times 500 \times 1000) = 500 + 5 \times 10^7$, corresponding to about 140 hours.

Tuple-based nested-loop join algorithm - iterator

Here is the iterator for the tuple-based nested-loop join:

```
Open(R,S) { R.Open(); S.Open(); s := S.GetNext();}
GetNext(R,S) {  REPEAT { r := R.GetNExt();
                         IF (NOT FOUND) { R.Close();
                                          s := S:GetNext();
                                          IF (NOT FOUND) RETURN;
                                          R.Open();
                                          r.GetNext();
                                        }
                       }
                UNTIL (the tuples r and s join);
                RETURN the tuple that is the join of r and s;
              }
Close(R,S) { R.Close(); S.Close(); }
```

Page-based nested-loop join algorithm

The tuple-based nested-loop join algorithm will never be used, because is highly inefficient. We improve it by considering the page-based nested-loop join algorithm.

2. Page-based nested-loop join

To compute the natural join of R(X,Y) and S(Y,Z)

we do as follows:

Scan the outer relation S, and for each page G of S:
Scan R, looking for the tuples "joining" those in G

The cost is B(S) + B(S) × B(R). In our example, the cost of the nested loop join is 500 + (1000 × 500) = 500.500, corresponding to about 1,5 hours.

Block nested-loop join algorithm

Suppose that the outer relation S fits in the buffer, and that the buffer has space for 2 extra frames. We can move the whole outer relation in the buffer, and use one of the extra frames for reading the inner relation R (one page at a time), and the other as output frame (so as to write the result in secondary storage one page at a time).

We come up with the one-pass algorithm that we already saw. The cost is B(R) + B(S), that is obviously optimal. In our example, the cost becomes 1500, corresponding to 15 seconds.

Block nested-loop join algorithm

In practice, it will be unlikely that the outer relation fits in the buffer, but it will be likely that we can use M frames of the buffer (plus 2 as extra frames, according to what we said before). We can then apply the idea of "block" nested-loop algorithm already discussed. The join algorithm in this case is:

3. Block nested-loop join

```
FOR each chunk of M pages of S DO
{ read the pages into the buffer and organize their tuples into a
  data structure whose search key is the common attributes of R and S;
  FOR each page b of R DO
  { read b in the buffer and for each tuple t of b use the data
    structure to find the tuples of S in the buffer joining with t;
    store in output the tuples that are the join of t with each of these tuples;
  }
}
```

The cost is B(S)+(B(R) × ceil(B(S)/M)), approximated as B(S) + B(R) × (1+B(S)/M). Obviously, the bigger is M, the more efficient the algorithm is. In our example, if we assume M=102, the cost is 500 + (1000 × (500/100)) = 5500, corresponding to about one minute.

Summary of block nested-loop algorithms

- B is the number of pages of the single operand
- R and S are the two operands, and B(R) and B(S) is the number of their pages (S is the smaller relation)
- We refer to the block nested-loop algorithms
- We assume that we have M+2 buffer frames free

Operator	# buffer frames M + 2 required	Page accesses (Disk I/O)
duplicate elimination	*any M > 1*	$B \times (3/2 + B/M) + M$
union, intersection, difference, cartesian product, join	*any M > 1*	$B(S) + B(R) \times (1 + B(S)/M)$

Two-pass algorithms

Two-pass algorithms can be seen as modifications of the one-pass algorithms, designed to cope with the situation where the relations are larger than what the one-pass algorithms can handle.

In two-pass algorithms, data are read from operands into main memory, processed in some way, written out to disk again, and then somehow re-read from disk to complete the operation.

Two-pass algorithms can be extended to multipass algorithms (like we did with merge sort), and will discuss such extensions later on.

We consider two types of two-pass algorithms:

1. based on sorting
2. based on hashing

Two-pass algorithms based on sorting

If we have a large relation R (or large relations R and S), such that B(R) (or, B(R)+B(S)) is greater than the available buffer frames M, then we can try to see if we can apply the two-pass algorithms based on sorting, constituted by the following two passes:

1. In the first pass, we repeatedly do the following (if we have two relations R and S, we do it both for R and for S):
   1.1) Read M pages of the relation into the buffer
   1.2) Sort these M blocks in main memory forming a so-called sublist, using an efficient sorting algorithm (we expect that the time to sort will not exceed the disk I/O time for step (1.1))
   1.3) Write the sorted sublist into M pages on secondary storage
2. The second pass processes the sorted sublists in some way to execute the desired operation (depending on the operation) and computing the result using one buffer frame for the output.

The schema is similar to two-pass external sorting, where sublists were called runs. As usual, we assume that whenever the output frame is full, it is written into the result in secondary storage, and we do not count the cost of such writes.

Two-pass algorithms based on sorting

- Duplicate elimination using sorting

  We have one relation R. After building the sublists for R, we use the available buffer frames to hold one page for each sublist (during the process, if a buffer frame becomes empty, we read another page from the same sublist), and we repeatedly copy (into the output buffer frame) the minimum tuple among the ones under considerations, ignoring all tuples identical to it.

  Space in buffer: In the second pass we need one output frame, and one frame for each sublist in the buffer, so in the first pass we cannot produce more than M-1 sublists, each M page long. So, it must be that ceil(B(R)/M) ≤ M-1 for executing the algorithm, which can be approximated as B(R)/M+1 ≤ M-1, i.e., B(R) ≤ M × (M-2), in turn approximated as B(R) ≤ $(M-1)^2$. In other words, the algorithm requires $\sqrt{B(R)}$ + 1 buffer frames.

  Cost of I/O: The number of page accesses performed by the algorithm (ignoring as usual the writing of the result) is 3B(R):

  1. B(R) to create the sublists, B(R) to write each of the sorted sublist
  2. B(R) to read each page from the sorted sublists

Duplicate elimination using sorting: example

Suppose two tuples (assumed to be integers) fit in each page and M=3, and the relation is:

2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3

After the first pass we have:

Sublist	Buffer	Disk (secondary storage)
R1		1 2, 2 2, 2 5
R2		2 3, 4 4, 4 5
R3		1 1, 2 3, 5

Duplicate elimination using sorting: example

Second pass:

Sublist	Buffer	Disk
R1	1 2	2 2, 2 5
R2	2 3	4 4, 4 5
R3	1 1	2 3, 5

We write 1 to the output, and we delete all occurrences of 1

Sublist	Buffer	Disk
R1	2	2 2, 2 5
R2	2 3	4 4, 4 5
R3	2 3	5

Duplicate elimination using sorting: example

We write 2 to the output, and we delete all occurrences of 2

Sublist	Buffer	Disk
R1	5	
R2	3	4 4, 4 5
R3	3	5

We write 3 to the output, and we delete all occurrences of 3

Sublist	Buffer	Disk
R1	5	
R2	4 4	4 5
R3	5	

Duplicate elimination using sorting: example

We write 4 to the output, and we delete all occurrences of 4

Sublist	Buffer	Disk
R1	5	
R2	5	
R3	5	

We write 5 to the output, and we delete all occurrences of 5

The final result is without duplicates: 1, 2, 3 ,4, 5 (written in 3 pages)

Two-pass algorithms based on sorting

- Grouping and aggregation using sorting

  In pass 1, we build the sublists for R, using as sort key the grouping attributes. In pass 2, we repeatedly do the following: find the least value of the sort key present among the first available tuples in the buffer. This value v becomes the next group, we prepare the corresponding tuple in the output frame, and we:

  (a) examine each of the tuples with sort key v and accumulate the needed aggregates (sum and count for avg)

  (b) during the process, if a buffer frame becomes empty, we read another page from the same sublist

  When there are no more tuples with sort key v, the tuple for the grouping value v is complete and when all the tuples in the output frames are complete, we consider it full, and we write it in secondary storage.

  Space in buffer: the algorithm will work as long as the number of sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) ≤ M-1, which means $B(R) \leq M \times (M-2)$ in turn approximated as $B(R) \leq (M-1)^2$. In other words, the algorithm requires $\sqrt{B(R)} + 1$ buffer frames.

  Cost of I/O: The number of page accesses performed by the algorithm (ignoring as usual the writing of the result) is 3B(R)

Two-pass algorithms based on sorting

- Set union using sorting

  The algorithm for computing $R \cup_S S$ is as follows:

(a)In pass 1, we build the sorted sublists for the two relations R and S.

(b)Use one buffer frame for each sublist of R and S, and repeatedly find the first remaining tuple t among all the frames. Copy t to the output frame and remove from the frames all copies of t (R and S are sets, and thus there are at most two copies). During the process, if a buffer becomes empty, then reload it with the next page from its sublist.

Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) + ceil(B(S)/M) ≤ M-1, approximated as B(R)/M + 1 + B(S)/M + 1 ≤ M-1. Thus, having $\sqrt{B(R) + B(S)} + 3$ buffer frames available is a sufficient condition for executing the algorithm.

Cost of I/O: the number of page accesses perfomed by the algorithm (ignoring as usual the writing of the result) is 3(B(R)+B(S))

Two-pass algorithms based on sorting

- Set intersection using sorting

The algorithm for computing $R \cap_S S$ is as follows:

(a) In pass 1, we build the sorted sublists for the two relations R and S.

(b) Use one buffer frame for each sublist of R and S, and repeatedly find the first remaining tuple t among all the frames. Copy t to the output frame if and only if it appears in both R and S, and remove from the frames all copies of t (R and S are sets, and therefore there are at most two copies). During the process, if a buffer becomes empty, then reload it with the next page from its sublist.

Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) + ceil(B(S)/M) ≤ M-1, approximated as B(R)/M + 1 + B(S)/M + 1 ≤ M-1. Thus, having $\sqrt{B(R) + B(S)} + 3$ buffer frames available is a sufficient condition for executing the algorithm.

Cost of I/O: the number of page accesses perfomed by the algorithm (ignoring as usual the writing of the result) is 3(B(R)+B(S))

Two-pass algorithms based on sorting

- Bag intersection using sorting

The algorithm for computing $R \cap_B S$ is as follows:

(a)In pass 1, we build the sorted sublists for the two relations R and S.

(b)Use one buffer frame for each sublist of R and S, and repeatedly find the first remaining tuple t among all the frames. Copy t to the output frame a number of times equal to the minimum between the number of times it appears in R and the number of times it appears in S, and remove from the frames all copies of t. During the process, if a buffer becomes empty, then reload it with the next page from its sublist.

Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) + ceil(B(S)/M) ≤ M-1, approximated as B(R)/M + 1 + B(S)/M + 1 ≤ M-1. Thus, having $\sqrt{B(R) + B(S)} + 3$ buffer frames available is a sufficient condition for executing the algorithm.

Cost of I/O: the number of page accesses perfomed by the algorithm (ignoring as usual the writing of the result) is 3(B(R)+B(S))

Two-pass algorithms based on sorting

- Set difference using sorting

The algorithm for computing $R -_S S$ is as follows:

(a)In pass 1, we build the sorted sublists for the two relations R and S.

(b)Use one buffer frame for each sublist of R and S, and repeatedly find the first remaining tuple t among all the frames. Copy t to the output frame if and only if it appears in R but not in S, and remove from the frames all copies of t (R and S are sets, and therefore there are at most two copies). During the process, if a buffer becomes empty, then reload it with the next page from its sublist.

Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) + ceil(B(S)/M) ≤ M-1, approximated as B(R)/M + 1 + B(S)/M + 1 ≤ M-1. Thus, having $\sqrt{B(R) + B(S)} + 3$ buffer frames available is a sufficient condition for executing the algorithm.

Cost of I/O: the number of page accesses perfomed by the algorithm (ignoring as usual the writing of the result) is 3(B(R)+B(S))

Two-pass algorithms based on sorting

- Bag difference using sorting

The algorithm for computing $R -_B S$ is as follows:

(a) In pass 1, we build the sorted sublists for the two relations R and S.

(b) Use one buffer frame for each sublist of R and S, and repeatedly find the first remaining tuple t among all the frames. Copy t to the output frame a number of times which is the difference between the number of times it appears in R and the number of times it appears in S, and remove from the frames all copies of t. During the process, if a buffer becomes empty, then reload it with the next page from its sublist.

Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as $\text{ceil}(B(R)/M) + \overline{\text{ceil}(B(S)/M)} \leq M-1$, approximated as $B(R)/M + 1 + B(S)/M + 1 \leq M-1$. Thus, having $\sqrt{B(R) + B(S)} + 3$ buffer frames available is a sufficient condition for executing the algorithm.

Cost of I/O: the number of page accesses perfomed by the algorithm (ignoring as usual the writing of the result) is $3(B(R)+B(S))$

Bag difference using sorting: example

Suppose two tuple (assumed to be integers) fit in each page and M=3, and the relations are:

R: 2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2

S: 1, 5, 2, 1, 3

After the first pass we have:

Sublist	Buffer	Disk (secondary storage)
R1		1 2, 2 2, 2 5
R2		2 3, 4 4, 4 5
S1		1 1, 2 3, 5

Bag difference using sorting: example

Second pass:

Sublist	Buffer	Disk
R1	1 2	2 2, 2 5
R2	2 3	4 4, 4 5
S1	1 1	2 3, 5

We do not write 1 to the output, and we delete all occurrences of 1

Sublist	Buffer	Disk
R1	2	2, 2, 2, 5
R2	2, 3	4, 4, 4, 5
S1	2, 3	5

Bag difference using sorting: example

We write 4 copies of 2 to the output, and we delete all occurrences of 2.

Sublist	Buffer	Disk
R1	5	
R2	3	4, 4, 4, 5
S1	3	5

We do not write 3 to the output, and we delete all occurrences of 3

Sublist	Buffer	Disk
R1	5	
R2	4, 4	4, 5
S1	5	

Bag difference using sorting: example

We write 3 copies of 4 to the output, and we delete all occurrences of 4

Sublist	Buffer	Disk
R1	5	
R2	5	
S1	5	

We write 5 once to the output, and we delete all occurrences of 5

The final result is: 2, 2, 2, 2, 4, 4, 4, 5 (written in 4 pages)

Two-pass algorithms based on sorting

- Simple sort-based join algorithm

This algorithm differs from the general pattern described for two-pass algorithms, since in the first pass we sort using Y the two relations that we want to join, R(X,Y) and S(Y,Z), through two-pass multi-way merge sort using M buffer frames (this means that $B(R) \leq M^2$ and $B(S) \leq M^2$).

In the second pass, we compute the join. In the first formulation, we assume that for no value y of Y the tuples of R and S with that value occupy more than M-1 buffer frames. If this is the case we repeatedly do the following (using one frame for R and one frame for S, and loading the frames with pages from R and S when necessary):

1. Find the least value y of the join attributes Y that is currently at the front of the frames for R and S.
2. If y appears only in one relation, then remove the tuples with y as Y-value, else identify all the tuples of both R and S with y as Y-value, using the other M-3 buffer frames, and put in the output frame all the tuples corresponding to the join of all the tuples of R with all the tuples of S with y as Y-value.

Two-pass algorithms based on sorting

- Simple sort-based join algorithm

If for some value y of Y the number of tuples of R and S with that value occupy more than M-1 buffer frames, then we have to modify the algorithm. In this second formulation, we do the following:

1. If the tuples from one relation, say R, that have y as Y-value fit in M-2 frames, then we load these pages of R into the frames, and read the pages of S that hold tuples with y, one at a time, into one frame. In other words, we are doing the one-pass join algorithm on only the tuples with y as Y-value.
2. If neither relations has sufficiently few tuples with y as Y-value, that they will fit in M-1 frames, then we use the M-1 frames to perform the nested-loop join of the tuples with Y-value y from both relations.

   In either case, it may be necessary to read pages from one relation, and ignore them for a later re-analysis. For example, in case (1), we might first read the pages of S that have tuples with Y-value y, and find out that there are too many to fit in M–1 frames. However, if we then read the tuples of R with that Y-value we might find that they do fit in M-1 frames.

Two-pass algorithms based on sorting

- Simple sort-based join algorithm

  Space in buffer: as we said before, in oder to be able to perform the sorting with the multi-way merge sort algorithm in two passes, it must be that the number of frames M is such that $B(R) \leq M^2$ and $B(S) \leq M^2$. In other words, the algorithm requires $\sqrt{\max(B(R),B(S))}$ buffer frames.

  Cost of I/O: since we have based our algorithm on two-pass multi-way merge sort using M buffer frames, the sorting phase requires 4(B(R) + B(S)) page accesses (one read and one write for each page in each of the two passes of the sorting algorithm). In the other phase, each page of R and S is read a fifth time. The exception would be if there were so many tuples with a common Y-value that we needed to do one of the specialized join on these tuples. In that case, the number of extra page accesses depends on whether one or both relations have so many tuples with a common Y-value that they require more than M frames by themselves. We will not go into all the details of these cases. We leave to the student as an exercise to perform the anaysis.

  So, assuming that no deviations from the process we have described are necessary (i.e., assuming the tuples with the same Y-value fit in M-1 frames of the buffer), the cost of the algorithm is 5(B(R) + B(S)).

Simple sort-based join algorithm: example

Let us analyze our example, where we consider the following values: B(R)=1000, B(S)=500. Also, we assume M = 102 buffer frames available, and 10ms per page access.

When we join the sorted sublists, we generally need only two of the 102 frames. However, if necessary, we could use 101 frames to hold the tuples of R and S sharing the same value of Y. Thus, to execute the algorithm, it is sufficient that for no value y the tuples sharing this value occupy more than 101 frames.

The cost is 5(B(R)+B(S)) = 7.500 page accesses, corresponding to one minute and 15 seconds.

Compared with the 5.500 page accesses of the block nested loop join algorithm, it seems that simple sort-based join algorithm is worse. However, if we recall that the cost of the block nested loop join algorithm is $B(S)+(B(R) \times (B(S)/(M-2) + 1))$, it is sufficient to consider a case where B(R)=10.000 and B(S)=5.000, to see that the block nested loop join algorithm would cost 515.000, while the simple sort-based join algorithm would cost 72.500.

Two-pass algorithms based on sorting

- Sort-merge join algorithm (or, sort-join algorithm)

  If we know in advance that the case of too many tuples with a common value of the join attributes does not occur or is rare, then we can use the following new algorithm: after the creation of the sublists, with sorting key Y, for both R and S, we

  1. Bring the first page of each sublist in the buffer (assuming that there are no more than M-1 sublists in total)
  2. Repeatedly find the least Y-value y among the available tuples of all the sublists. Identify all the tuples of both relations that have Y-value y, perhaps using some of the M available frames, if there are fewer than M sublists. Put in the output frame the join of all tuples from R with all the tuples from S sharing this common Y-value. If the frame of one of the sublists is exhausted, then reload it with the next page from its sublist.

  Space in buffer: the algorithm will work as long as the sorted sublists created at pass 1 is less than or equal to M-1, i.e., as long as ceil(B(R)/M) + ceil(B(S)/M) ≤ M-1, approximated as B(R)/M + 1 + B(S)/M + 1 ≤ M-1. Thus, having $\sqrt{B(R) + B(S)}$ + 3 buffer frames available is a sufficient condition for executing the algorithm.

  Cost of I/O: for each page, we have two accesses to create the sublists, and one access in the subsequent "join" phase; thus the cost is 3(B(R) + B(S)).

Sort-merge join algorithm: example

Let us consider our example, where we have the following values: B(R)=1000, B(S)=500. Also, we assume M = 102 buffer frames available, and 10ms per page access.

Suppose we divide R into 10 sorted sublists, and S into 5 sorted sublists, each of length 100. We then use 15 frames to hold the current page of each of the sublists. If many tuples have a fixed Y-value, then we can use the remaining frames to store these tuples. Assuming that this is sufficient, the total number of page accesses is 3(1.000 + 500) = 4.500, corresponding to 45 seconds.

Sort-merge join algorithm: observations

The case of many tuples having a fixed Y-value represents a problematic situation for the sort-merge join algorithm. The following observations deal with this problem.

1. Sometimes we are actually sure that the problem will not arise. For example if Y is a key for R, then a given Y-value y can appear only once among all the pages of the sublists for R. When it is y's turn, we can leave the tuple from R in place and join it with all the tuples of S that match. If pages of S's sublists are exhausted during this process, they can have their frames reloaded with the next page, and there is never any need of additional space, no matter how many tuples of S have Y-value y.
2. If M is much greater than $\sqrt{B(R) + B(S)}$, we shall have many unused frames for storing tuples with a common Y-value.
3. If all else fails, we can use a nested loop join algorithm on just the tuples with a common Y-value, using extra page accesses, but getting the job done correctly.

Summary of two-pass algorithms based on sorting

- B is the number of pages of the single operand
- R and S are the two operands, and B(R), B(S) are the number of their pages

Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$\sqrt{B} + 1$	$3B$
union, intersection, difference	$\sqrt{B(R) + B(S)} + 3$	$3(B(R) + B(S))$
simple sort join	$\sqrt{max(B(R), B(S))}$	$5(B(R) + B(S))$
sort-merge join	$\sqrt{B(R) + B(S)} + 3$	$3(B(R) + B(S))$

Two-pass algorithms based on hashing

The idea of two-pass algorithms based on hashing is the following:

1. If the data is too big to be processed in one-pass, hash all the tuples of the operand (or the operands) using an appropriate hash key and function.
2. For all the common operations, there is a way to select the hash key so that all the tuples that need to be considered together when we perform the operation has the same hash value computed by the hash function, and therefore are in the same bucket.
3. We then perform the operation by working on one bucket at a time (or on a pair of buckets with the same hash value, in the case of binary operation).

If there are M buffer frames available, we can pick M or M-1 as the number of buckets, thus gaining a factor of M in the size of relations we can handle with respect to the case of the one pass algorithms.

Partitioning relations by hashing

Let us see how we can take a relation R and, using M buffers, partition R into M-1 buckets in secondary storage of about equal size. The following partitioning algorithm uses h as a hash function:

```
initialize M-1 buckets using M-1 empty frames
FOR each page b of relation R DO
BEGIN  read page b into the Mth frame
       FOR each tuple t in b DO BEGIN
              IF the frame for bucket h(t) has no room for t THEN
                     BEGIN copy the frame to secondary storage;
                            initialize a new empty page in that frame
                     END
              copy t to the frame for bucket h(t)
       END
END
FOR each bucket DO IF the corresponding frame is not empty
                     THEN write the frame to secondary storage
```

Two-pass algorithms based on hashing

- Duplicate elimination using hashing

  After partitioning the relation using the above partitioning algorithm (where the hash function works on all the attributes), we can carry out the second pass. Indeed, since two copies of the same tuple t will hash to the same bucket, we can examine one bucket at a time in isolation, and produce R_i, the portion of R that hashes to the i-th bucket. Then we can take as the answer the union of such R_i. When we analyze the single bucket in isolation, we can use the one-pass algorithm for duplicate elimination (this requires that each R_i is sufficiently small to fit in the buffer).

  Space in buffer: Since we assume that h partitions R into equal sized buckets, each Ri will be constituted by approximately B(R)/(M-1) pages. The overall two-pass algorithm works if this number is no larger than M, i.e., if $B(R) \leq M(M-1)$, approximated as $B(R) \leq (M-1)^2$.

  Cost of I/O: for each page, we have one access for reading and one acess for writing when we hash the tuples. We then read each bucket again in the one-pass algorithm that focuses on that bucket. Therefore, the cost is 3B(R).

Two-pass algorithms based on hashing

- Grouping and aggregation using hashing

  We partition the relation by using a hash function that depends only on the grouping attributes. After that, we can carry out the second pass, where we use the one-pass algorithm to process each bucket in turn. When we analyze a bucket “b”, we form the tuple of the result derived from each group stored in “b”, knowing that all such tuples are stored in such bucket.

  Space in buffer: We can surely process each bucket in the buffer if $B(R)/(M-1) \leq M-1$, where M is the number of available buffer frames. However, in the second pass, we only need one tuple per group as we process each bucket. Thus, even if the size of a bucket is larger than M, we can handle the bucket in one pass, provided the tuples for all the groups in the bucket require no more than M-1 buffer pages. It is thus sufficient that the pages needed to store all groups in each bucket is less than or equal to M-1. If the number of pages of each bucket is $B(R)/(M-1)$, then the pages needed for storing one tuple per group in each bucket is $(B(R)/(M-1))/A$, where A is the average number of tuples per group. Since $(B(R)/(M-1))/A$ must be less than or equal to M, a more accurate upper bound is $B(R) \leq (M-1)(M-1)A$.

  Cost of I/O: analogously to the duplicate elimination algorithm, for each page, we have one access for reading and one access for writing when we hash the tuples. We then read each bucket again in the one-pass algorithm that focuses on that bucket. Therefore, the cost is $3B(R)$.

Two-pass algorithms based on hashing

- Union, intersection and difference using hashing

  We partition each operand R and S relation by using one hash function. Both R and S will have M-1 buckets after the first pass (overall, we will have 2(M-1) buckets). We then load in the buffer the various pairs of buckets (at each step, bucket R_i and bucket S_i will be considered together, as they agree on the value of the hash function), and we compute the result for this bucket by means of the appropriate one-pass algorithm, depending on the operation we have to execute.

  Space in buffer: we must be able to take the one-pass union, intersection, or difference of R_i and S_i whose size will be approximately B(R)/(M-1) and B(S)/(M-1). Recall that the one-pass algorithms for these operations require that the smaller operand is in at most M-2 pages. Thus, the two-pass hash-based algorithms require $\min(B(R),B(S)) \leq (M-1)(M-2) \leq (M-2)^2$.

  Cost of I/O: we should remember that the one-pass algorithms for union, intersection and difference, require B(R)+B(S) page accesses; to this, we must add the two accesses per page during the first pass. The total is 3(B(R)+B(S)) page accesses.

Two-pass algorithms based on hashing

- Join using hashing: the hash-join algorithm

  Like in the case of the other binary operations, we partition each operand R and S relation by using one hash function. Together, R and S will have M-1 buckets after the first pass. We then load in the buffer each pair of buckets in turn (at each step, bucket R_i and bucket S_i, agreeing on the value of the hash function, will be considered together), and we compute the join for this bucket, by using the one-pass join algorithm.

  Space in buffer: we must be able to take the one-pass join of R_i and S_i whose size will be approximately B(R)/(M-1) and B(S)/(M-1), respectively. Recall that the one-pass algorithms for these operations require that the smaller operand occupies at most M-2 pages. Thus, the two-pass hash-based algorithms require $\min(B(R),B(S)) \leq (M-1)(M-2) \leq (M-2)^2$.

  Cost of I/O: we should remember that the one-pass algorithm for join requires B(R)+B(S) page accesses; to this, we must add the two accesses per page during the first pass. The total is 3(B(R)+B(S)) page accesses.

Hash-join algorithm: example

Let us analyze our example, where we consider the following values: B(R)=1000, B(S)=500. Also, we assume M = 102 buffer frames available, and 10ms per page access.

We may hash each relation to 100 buckets, so the average size of a bucket is 10 blocks for R and 5 blocks for S. Since the smaller number, i.e., 5, is much less than the number of available frames, we expect to have no trouble performing a one-pass join on each pair of buckets.

The cost is 1500 page accesses for reading the two relationsh while hashing into buckets, another 1500 to write all the buckets, and a third 1500 to read the pages during the one-pass join phase. The total is 4500, corresponding to 45 seconds.

Hybrid hash-join algorithm

Suppose that B(S) << (M-1)(M-2). Why using M-1 buckets, when we could use less?

- Partition S into k buckets (k < M-1)
  - m buckets S_1 , …, S_m stay in memory (m < k)
  - (k-m) buckets S_{m+1}, …, S_k go to disk, as before
- Partition R into k buckets
  - m buckets join immediately with S_1 , …, S_m
  - (k-m) buckets go to disk, as before
- Finally, join k-m pairs of buckets:

  (R_{m+1},S_{m+1}), (R_{m+2},S_{m+2}), …, (R_k,S_k)

Hybrid hash-join algorithm

More precisely, suppose we choose to use k (<< M-1) buckets.

In the first phase, we hash S and R. When we hash S, we keep m buckets $S_1,\ldots,S_m$ in memory (m < k), and we treat the other (k-m) buckets $S_{m+1},S_{m+2},\ldots, S_k$ as before. We then read R one page at a time in an input frame, and we use (k-m) buckets.

- If a tuple t of R hashes to one of the first m buckets, then we immediately compute the join with the tuples of the corresponding bucket in main memory, using one output frame for storing the result of such join.
- If t hashes to one of the buckets whose corresponding S-bucket is on disk, then t is sent to the buffer frame for the right bucket among Rm+1,Rm+2,…, Rk , and eventually migrates to disk, as before.

In the second phase, we compute the join of the (k-m) pairs of buckets (R_{m+1},S_{m+1}), (R_{m+2},S_{m+2}), …, (R_k,S_k), using the one-pass technique.

Hybrid hash-join algorithm

- We can use the above idea provided the expected size of the buckets in the buffer, plus one frame for each of the other buckets, plus one frame for reading R and one frame for the output does not exceed M:

$$m \times B(S)/k + k - m + 2 \leq M$$

- How to choose k and m ?
  - Choose k s.t. k < M - 1
  - Choose m/k large but s.t. $m/k \times B(S) \leq M$
- Assuming $m/k \times B(S) >> k - m$, we can choose $m/k = M/B(S)$
- A very popular choice, coherent with the above considerations, is m=1 and k = B(S) / M.

Hybrid hash-join algorithm

How many page accesses for the hybrid hash-join algorithm?

- Since hybrid join saves 2 page accesses for the buckets of S that remain in memory and the corresponding R-buckets, we conclude that the saving is a m/k fraction of B(R) + B(S).
- In other words, hybrid join saves $2 \times m/k \times (B(R) + B(S))$ page accesses.
- This means that if m=1 and k = B(S) / M, then we save $2 \times M/B(S) \times (B(R) + B(S))$, and the cost is:

$$(3-2m/k) \times (B(R) + B(S)) = (3-2M/B(S)) \times (B(R) + B(S))$$

Summary of two-pass algorithms based on hashing

- B is the number of pages of the single operand
- R and S are the two operands, and B(R), B(S) are the number of their pages
- In the last row, we assume that B(S) ≤ B(R)

Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$\sqrt{B} + 1$	$3B$
union, intersection, difference	$\sqrt{min(B(R),B(S))} + 2$	$3(B(R) + B(S))$
hash-join	$\sqrt{min(B(R),B(S))} + 2$	$3(B(R) + B(S))$
hybrid hash-join	$>> \sqrt{min(B(R),B(S))} + 2$	$(3-2M/B(S)) \times (B(R) + B(S))$

Comparison between sort- and hash-based algorithms

1. Hash-based algorithms for binary operations have a buffer requirement that depends only on the smaller of the two arguments rather than on the sum of the argument sizes, as for sort-based algorithms
2. Sort-based algorithms in principle allow us to produce a result in sorted order, and we can take advantage of that sorting later. The result might be used by a sort-based algorithm later (in the course of the execution of the same query, for example), or it could be the answer to a query that is required to be produced sorted. However, since we have to execute the sorting algorithm, the operations in main memory are more costly.
3. Hash-based algorithms depend on the buckets being of equal size, in theory. Since in practice there is generally at least a small variation in size, it is not possible to use buckets that, on average, occupy M pages. We must limit them to a somewhat smaller figure. This is especially prominent if the number of different hash values is small, e.g., performing a group-by on a relation with few groups or a join with few values for the join attributes.

Exercise 4

Consider the binary operators and tell whether, for such operators, there are conditions under which the block nested-loop is better than the two-pass algorithms in terms of number of page accesses.

Index-based algorithms

Index-based algorithms are algorithms that make use of indexes. We will see that there are index-based algorithm that use one-pass, and others that use a nested-loop schema.

We introduce the notion of conformance of an index to a condition, that will be useful in our discussion. Intuitively, an index conforms to a condition C when it can be used effectively to evaluate C. A simple index is said to conform to

attr op value

if

- it is a tree index, the search key is attr, and op is <, <=, =, <>, >=, or >
- it is a hash index, the search key is attr, and op is =

Prefix of a search key: initial non-empty segment of attributes for a composite search key

Example: for a search key <a,b,c>, <a> and <a,b> are prefix, while <a,c> and <b,c> are not prefix

Conformance of an index to a condition

An index (either simple or composite) is said to conform to the conjunction

(att op value) and (att op value) and …

if

- the index is tree-based, and there exists a prefix P of the search key such that, for each attribute att in P, there is a term of the form (att op value) in the conjunction (such terms are said the primary terms of the conjunction)
- the index is a hash index, and for each attribute of the search key there is a term (att = value) in the conjunction.

Examples

- A hash index with search key <r,b,s> conforms to the condition (r = 'J' and b = 5 and s = 3), not to (r = 'J' and b = 7)
- A tree index with search key <r,b,s> conforms to the condition (r > 'J' and b = 7 and s = 5), and to (r = 'H' and s = 9), not to (b = 8 and s = 10)
- Any index with search key <b,s> conforms to the condition (d > 1000 and b = 5 and s = 7): after finding the entries for the primary terms b = 5 and s = 7, we select only those satisfying the further condition d > 1000
- If we have an index with search key <b,s> and a tree index with search key d, then both conform to the condition (d > 1000 and b = 5 and s = 7). No matter which one is used, we will select only the tuples retrieved with the help of the index that satisfy the further condition

Index-based selection

Index-based algorithms for selection are special one-pass algorithms, that are in general even more efficient than classical one-pass algorithms.

We consider the query

```
select *
from R
where <att op value>
```

and distinguish between various cases, depending on the method used to represent R. As usual, we will express the cost in terms of page accesses. We will assume alternative 2, if not otherwise stated. Also, remember that we assume that relations are clustered (if not otherwise stated).

Index-based selection

Case 1) The attribute is a key of the relation, op is =, and we have an index on such attribute

If the index is a hash index, then the index conforms to the condition, and we can use the index. The cost of the selection operation is 1 (or 2, if we count one access for the bucket, and one access for one overflow page).

If the index is a tree-based index, then the index conforms to the condition, and we can use the index. We refer to the cost of equality search for tree-based index. If we do not know the number of pages in the leaves, or the fan-out of the tree, then we can assume that the cost is 3-4 page accesses.

Note that the estimation of 3-4 page accesses is a realistic upper bound, because with a fan-out of 100, this corresponds to a range of 100.000 – 100.000.000 leaves (if 10 data entries fit in one page, this corresponds to a range of 1.500.000 – 15.000.000.000 data records in the data file). The estimation becomes 2-3 if the root fits in the buffer.

Index-based selection

Case 2) The attribute is not a key of the relation, op is =, and we have a weakly clustering hash index on such attribute

The fact that the index is weakly clustering means that, given a value v for the attribute, the tuples with that value will be found in as few pages as possible (typically, one or two).

Let us denote with T(R) the number of tuples of R, and with V(R,A) the number of distinct tuples in the projection of R on attribute A (note that V(R,A)=T(R) if A is a key of R). If the = condition is on attribute A, and there is a hash index on A, then the index conforms to the condition. The cost of the selection operation is 1 + B(R)/V(R,A) or 2 + B(R)/V(R,A). Indeed, the average size of each page is T(R)/B(R), and the average number of tuples with the same value of A is T(R)/V(R,A), and therefore the average number of pages that we have to access for accessing all the tuples with a given A-value is (T(R)/V(R,A)) / (T(R)/B(R)) = B(R)/V(R,A). In practice, the cost will be somehow higher, because the tuples with A=v may be spread over several pages (for example, if the pages of R has room for the growth of R, or R is not a clustered relation). In other words, B(R)/V(R,A) is the number of pages required to hold the tuples with a certain value for A.

Index-based selection

Case 3) The attribute is not a key of the relation, op is =, and we have a clustering tree index on such attribute

The fact that the index is clustering means that the data file is ordered with the same criterion as the index.

Again, all the records satisfying the condition can be found in as few pages as possible (typically, one or two).

The index again conforms to the condition, and thus we can use the index. As we saw in our analysis of the tree-based index, we have to compute the number B of required leaf pages. Then the average cost of the selection operation is $\log_F 1.5B + B(R)/V(R,A)$. As usual, if we do not know the number of pages in the leaves, or the fan-out of the tree, then we can assume that the cost is 3-4 page accesses plus B(R)/V(R,A).

Index-based selection

Case 4) The attribute is not a key of the relation, op is =, and we have a nonweakly clustering index on such attribute

Since the index is not weakly clustering, then each index entry can point to a qualifying record on a different page, and the cost could be one page per qualifying record. We can do better by first sorting the rid in the index data entries by their page-id component, so that when we bring in memory a page of R, all qualifying records in this page are retrieved one after the other. The cost of retrieving the qualifying records is now the number of pages of R that contain qualifying records.

If we assume that 10% of records of R qualify, in particular, we assume that 1.000 tuples in 100 pages are qualifying over the whole 1000 pages, then the cost for clustered tree index is 3-4 plus 100, whereas the cost for the nonweakly clustering tree index is 3-4 plus 1.000 pages (note that scan costs 1000!). If the index is nonweakly clustering, and we sort the rids, then things get better, but it is likely that the qualifying records are stored in much more than 100 pages (therefore, the cost is higher than in the clustered case).

Index-based selection

Case 5) The condition involves one attribute, and op is > or <, or is a range equality, and we have a clustering tree index on such attribute

The index conforms to the condition, and thus we can use the index. We refer the reader to the analysis that we carried out in the study of the tree-based index.

Case 6) The condition is a complex condition

A selection with a complex condition *C* can sometimes be implemented by splitting the condition into atomic condiitons, and using index-based selections for atomic conditions if possibile (possibly followed by another selection on only those tuples retrieved by means of the index). For example, if *C* is of the form a=v AND *C'*, then we can split the selection into a cascade of two selections, the first checking only for A=v, and the second checking condition *C'*. The first is a candidate for an index-based selection.

Examples of costs of index-based selection

Suppose B(R)=1000, and T(R)=20.000, i.e., R has 20.000 tuples that are packed 20 to a block. Let A be one of the attributes of R, and assume that there is an index on A. Consider the selection of R with condition A=0.

1. If V(R,A)=100, and the index is clustering, then the index-based selection requires the page accesses for the index (probably 2), plus 1000/100=10 page accesses.
2. If V(R,A)=100, and the index is non clustering, then the index-based selection requires the page accesses for the index (probably 2), plus more index pages, plus 20.000/100=2000 data page accesses. Note that this cost is higher than scanning the entire relation.
3. If A is a key, i.e., V(R,A)=20.000, then the index-based selection requires the page accesses for the index (probably 2), plus 1 page accesses, regardless whether the index is clustering or not.

Index-based projection: “index-only scan”

If we have to compute the projection on A1,…,An of relation R, and we have a dense index (in particular, a B^+-tree index) whose search key includes all attributes A1,…,An, then we can use the index, in particular for the so-called “index-only scan”.

An index-only scan for a tree-based index is a scan of the leaves of the tree, that is much more efficient than the scan of the data file (because the data file is usually much bigger than the collection of the leaf pages)

If A1,…,An form a prefix of the search key, then, during the index-only scan we can efficiently (see one-pass algorithm for sorted relation):

- eliminate those attributes that are not among the target attributes in the projection
- eliminate duplicates, if needed (relying on the fact that the leaves are ordered)

Other index-based algorithms

As we said before, the above mentioned index-based algorithms for selection and projections can also be seen as special cases of one-pass algorithms.

In addition, all the following operations:

- duplicate elimination
- aggregation
- union, intersection, difference,
- join

can be profitably carried out with the help of an index (if the index is a suitable one).

We only describe the join operation in what follows, and leave the other operations as an exercise. The index-based join algorithms can be seen as special nested-loop join algorithms.

Exercise 5

Define suitable index-based algorithms for the following operations:

- duplicate elimination
- aggregation
- union, intersection, difference.

Index-based join

The first case of index-based join is the so-called index-nested loop algorithm. Suppose we want to compute the natural join between R(X,Y) and S(Y,Z) , and S has an index on Y. The algorithm considers one page of R at a time, and within each page, for each tuple t it uses the index to find all tuples of S having t[Y] as Y-value, thus outputting the join of each of these tuples with t. This is a special case of the tuple-based nested loop algorithm, where the access to S is supported by the index.
The cost is B(R) plus T(R) times the cost of accessing S through the index. For each access to S, i.e., for each tuple t of R, we have the cost of accessing the index (and this will depend on the index), plus the cost of accessing the set P of pages of S needed to retrieve the T(S)/V(S,Y) tuples with the value for Y given by t.

- If the index is non clustering, then P has T(S) / V(S,Y) pages in the worst case.
- If the index is clustering, in the average case P has (T(S) / V(S,Y)) / N pages, where N is the average number of tuples of S per page, i.e., T(S)/B(S). So, for clustering index, the number of page accesses to S for each tuple t of R is B(S) / V(S,Y) and the cost of the join is B(R) + T(R) × (cost of using the index for one value +B(S)/V(S,Y)).

Index-based join

Coming back to our example (R with 1000 pages and S with 500 pages), assume that 10 tuples of either relation fit in one page, so T(R)=10.000 and T(S)=5000. Assume also that V(S,Y)=100, i.e., there are 100 different values of Y in the tuples of S, and suppose that we have a clustering tree index on Y for S, for which the cost of equality search is 2. Then the cost of the index-based join is 1000 + 10.000 × (2 + 500 / 100) = 70.000.

Note that this number is high with respect to the other join algorithms. However, there are situations where the index-based algorithm is advantageous. For example, if the relation R is very small compared to S, and V(S,Y) is high (for example, when Y is a key for S), then we access only a fraction of the pages of S, since most of Y-values will not appear in R at all (conversely, both sort-based and hash-based join methods will examine every tuple of S at least one).

Index-based join

The second case of index-based join we consider is the natural join between R(X,Y) and S(Y,Z) under the assumption that one of the two indexes is sorted, or both of them are sorted.

Let us consider the case where we have one sorted index on Y, either for R or for S. We can perform the ordinary sort-join algorithms, but with the advantage of avoiding the sorting of one of the relation, the one for which we have the sorted index, and with the advantage of working on the index, and accessing the data files only when necessary.

If we have one sorted index for each relation, then we perform only the final step of the simple sort-based join algorithm, thus obtaining the so-called zig-zag join, whose name comes from the fact that we jump back and forth between the indexes (NOT the relations) finding Y-values that they share in common.

Notice that the tuples of a relation with a Y-value that does not appear in the other relation need never be retrieved.

Index-based join

In the case of sorted clustering index, the retrieval of all tuples of the corresponding relation with a given search key value will result in a number of page accesses proportional to the number of pages of the relation needed to store such tuples.

In the case of nonclustering sorted index, the retrieval of all tuples of the corresponding relation of a given search key value may result in a number of page accesses proportional to the number of such tuples.

Note that, if for a given value of the search key, there are so many tuples from R and S that neither fits in the buffer, we have to modify the algorithm similarly to what we did in the second formulation of the sort-based join algorithm. However, this situation will be not the typical one. On the contrary, in the typical case the step of joining all tuples with a common Y-value will be carried out with only as many page accesses as it takes to read them.

Index-based join

Coming back to our example (R with 1000 pages and S with 500 pages), assume that there is a clustering sorted index on Y for S, and no index for R.

Assuming 101 available buffer frames, we may use them to create 10 sorted sublists for the 1.000 pages of R, with the cost of 2.000. We then use 11 frames, 10 for the sublists of R and one for the tuples of S, retrieved via the index. If we have 50 pages for the leaves of the tree, then the total cost is 2.000 (for sorting R) + 1.000 (for reading R after sorting) + 50 (for reading the leaves of the tree) + 500 (for reading S). The total is 3.550.

If both R and S have a clustering sorted index on Y, then we avoid sorting R, and we use just 1.550 page accesses.

Multipass algorithms

As we already said, the two-pass algorithms we have discussed can be used when there is a limit to the size of the relations representing the operands.

In fact, these algorithms can be generalized to algorithms that, using as many passes as necessary, can process relations of arbitrary size.

Following the structure of two-pass algorithms, we consider the generalizations of two kinds:

1. sort-based,
2. hash-based.

Remainder: Multipass sort-merge algorithm

Two formulations: iterative and recursive.

Recursive formulation:

- Base step: If R fits in the M frames available in the buffer (i.e., B(R) ≤ M), then sort R in the buffer, using any main memory algorithm and write the sorted relation to secondary storage
- Inductive step: If R does not fit in the M frames available of the buffer, partition the pages of R into M-1 groups $R_1,...,R_{M-1}$, and recursively sort R_i for each i=1,2,...,M-1. Then merge the M-1 sorted sublists (runs) using M-1 buffer frame as input and one buffer frame for the output, and write the sorted relation to secondary storage.

Multipass sort-based algorithms

To describe the structure of a multipass sort-based algorithm we distinguish between unary and binary operations.

Let us first analyze the case of unary operations, assuming that there are M available buffer frames.

1. First phase (partition/sorting phase): We partition R into M-1 parts, sort each of them using the (multipass) merge-sort algorithm,
2. Second phase: we load one page at a time of the sorted sublists of R, performing the operation on the tuples at the front of the sublists using a buffer frame for the output. In particular:
   - For duplicate elimination (for which the sorting phase was done on all the attributes) we output one copy for each distinct tuples
   - For aggregation (for which the sorting phase was done only on the grouping attributes), we combine the tuples with a given value of the grouping attributes in the appropriate manner, depending on the aggregation function.

Multipass sort-based algorithms

In the case of binary operations, we essentially use the same idea.

1. First phase (partition/sorting phase): the partition into M-1 parts is done on the two relations R and S in such a way that R is divided into M_R parts, and S in M_S parts such that $M_R + M_S <= M-1$. Each part is sorted using the (multipass) merge-sort algorithm, so as to produce the sorted sublists
2. Second phase: we read each pair of corresponding sorted sublists (we have at most M-1 pairs), and we operate in the buffer using one output buffer frame, and we do the computation depending on the operation.

In the first phase, we can divide the M-1 buffer frames between relations R and S as we wish. One method for minimizing the total number of passes, is to divide the buffer frames in proportion to the number of pages of the two relations. That is, R gets $M_R=(M-1) \times B(R) / (B(R) + B(S))$ frames at most, and S gets the rest.

Multipass sort-based algorithms

If K is the number of passes of the algorithm, then we perform K-1 passes in the partition/sorting phase, and one final pass in the second phase. As we saw when we studied the multipass merge-sort algorithm, at each step of the first phase, we read and write all the pages (and we also count the writing of the result). In the second phase, we read all the pages (as usual, we ignore the cost of writing the final result). Therefore the cost is:
- for unary operations, $2(K-1)\,B(R) + B(R) = (2K-1)\,B(R)$,
- for binary operations, $2(K-1)(B(R)+B(S))+B(R)+B(S) = (2K-1)(B(R)+B(S))$.

In order to perform a unary operation in K passes, we need M buffer frames such that $B(R) \leq (M-1)^{K-1} \times M$, that means that $B(R) \leq (M-1)^K$ is a sufficient condition for performing the operation. In other words, we need $B(R)^{1/K} + 1$ buffer frames.

In order to perform a binary operation in K passes, we need M buffer frames such that $B(R) + B(S) \leq (M-1)^{K-1} \times M$, that means that $B(R) + B(S) \leq (M-1)^K$ is a sufficient condition for performing the operation. In other words, we need $(B(R)+B(S))^{1/K} + 1$ buffer frames.

Summary of multipass sort-based algorithms

- B is the number of pages of the single operand
- R and S are the two operands, and B(X) is the number of pages for X
- K is the number of passes

Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$B^{1/K} + 1$	$(2K-1)B$
union, intersection, difference, join	$(B(R)+B(S))^{1/K} + 1$	$(2K-1)(B(R)+B(S))$

Multipass hash-based algorithms

We hash the relation or the relations into M-1 buckets, where M is the number of available buffer frames. We then apply the operation on each bucket individually, in the case of unary operations. If the operation is binary, then we apply the operation to each pair of corresponding buckets. The result of the operation on the entire relation will be the union of the results on the buckets. We can describe this approach recursively:

Basic step: for a unary operation, if the relation fits in M buffer frames, read it and perform the operation. For a binary operation, if either relation fits in M-1 buffer frames, perform the operation by reading this relation into buffer and then read the second relation one page at a time, into the Mth buffer frame.

Inductive step: If no relation fits in the buffer, then hash each relation into M-1 buckets, and recursively perform the operation on each bucket (unary operation) or corresponding pair of buckets (binary operation), and accumulate the output from each bucket or pair.

Multipass hash-based algorithms

Note that, obviously, at every pass we use a different hash function.

We make the assumption that when we hash a relation, the tuples distribute as evenly as possible among the buckets. The cost is analogous to the multipass sort-based algorithm:

- for unary operations, (2K-1)B(R),
- for binary operations, (2K-1)(B(R)+B(S)).

Consider a unary operation. Let u(M,K) be the number of pages in the largest relation that a K-pass hash-based algorithm can handle. We can define u recursively by:

Basis: u(M,1)=M-1, i.e., B(R) ≤ M-1.

Induction: We assume that the first step divides the relation into M-1 buckets of equal size. The buckets for the next pass must be sufficiently small that they can be handled in K-1 passes; that is, the buckets are of size u(M,K-1). Since R is divided into M-1 buckets, we must have $u(M,K)=(M-1) \times u(M,K-1)$. This implies $u(M,K)=M(M-1)^{k-1}$, that we can approximate with $u(M,K)=(M-1)^k$. This means that we can perform one of the unary operations on R in K passes with M buffer frames if $B(R)^{1/K} + 1 \leq M$.

Multipass hash-based algorithms

We can perform a similar analysis for binary operations. Let us consider the join, as an example. Let j(M,K) be an upper bound on the size of the smaller of the two relations R and S involved in the natural join between R(X,Y) and S(Y,Z).

Basis: j(M,1)=M-1, because if we use the one-pass algorithm for the join, then either R or S must fit in M-1 pages.

Induction: $j(M,K)=(M-1) \times j(M,K-1)$, since on the first of K passes, we divide each relation into M-1 buckets, and we may expect each bucket to be 1/(M-1) of its entire relation, but we must then be able to join each pair of corresponding buckets in K-1 passes.

We can conclude that $j(M,K)=(M-1)^k$. That is, we can join R and S using K passes and M buffer frames, if $\min(B(R),B(S))^{1/K} + 1 \leq M$.

Summary of multipass hash-based algorithms

- B is the number of pages of the single operand
- R and S are the two operands, and B(X) is the number of pages for X

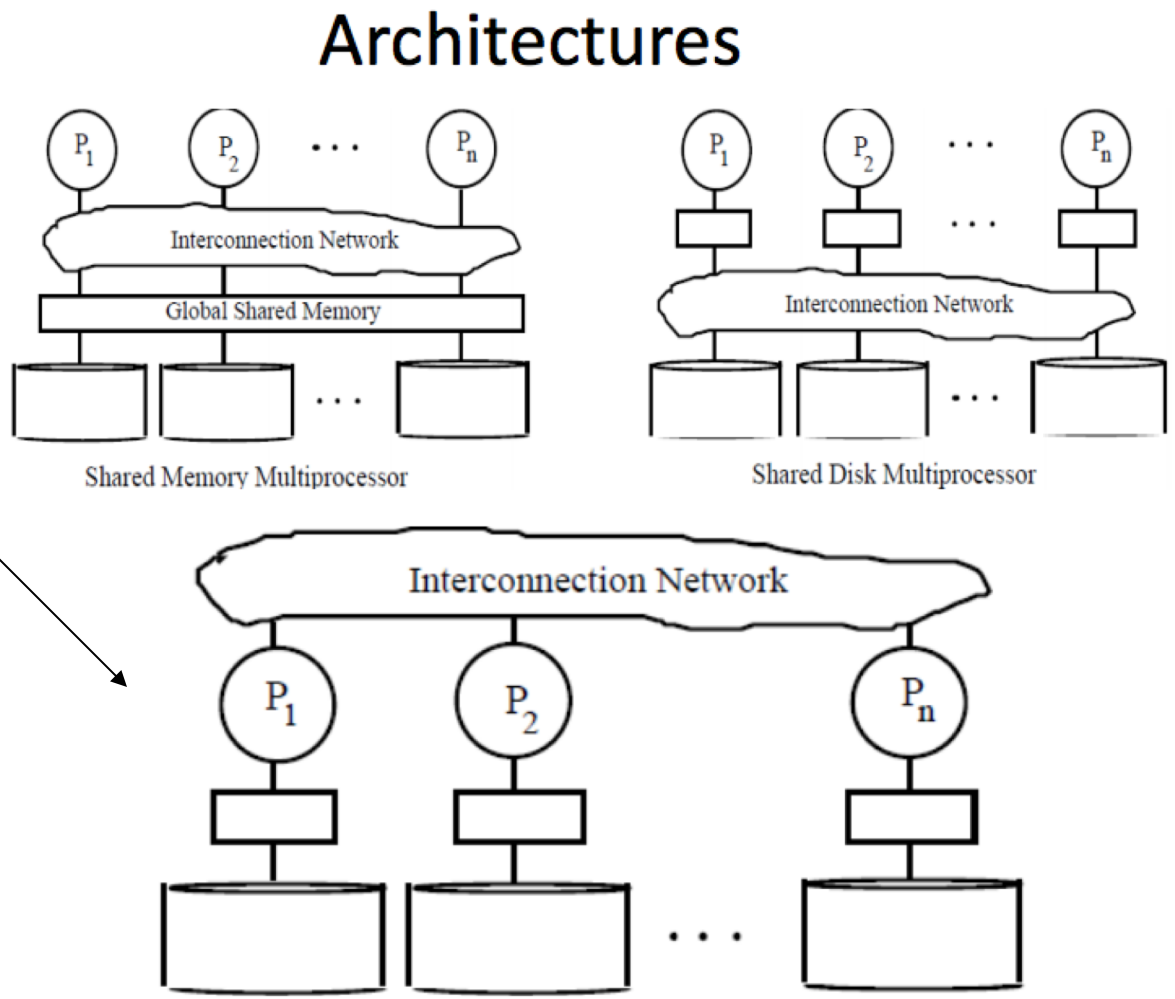
Operator	# buffer frames M required	Page accesses (Disk I/O)
duplicate elimination, grouping	$B^{1/K} + 1$	$(2K-1)B$
union, intersection, difference, join	$min(B(R),B(S))^{1/K} + 1$	$(2K-1)(B(R)+B(S))$

Exercise 5

Consider the binary operators and tell whether, for such operators, there are conditions under which the block nested-loop is better than the 3-pass algorithms in terms of number of page accesses.

Parallel algorithms

We refer to the "shared-nothing" architecture, where all processors have their own memory and their own disks (secondary storage) and do not share any resource.



Parallel algorithms

In the “shared-nothing” architecture, the communications from processor to processor is via the communication network.

For example, if one processor wants to read tuples from the disk of another processor, it sends a message asking for that data and the processor sends the data through the network.

We assume that the communication cost is much less than the cost of reading/writing a page from/in secondary storage. So, we often neglect such cost.

Horizontal data partioning

Parallel evaluation in relational algebra is based on the fact that a relation R is split into P chunks $R_0,...,R_{P-1}$, stored at the P nodes.

Strategies for partitioning a relation R:

- Round robin: tuple t_i goes to chunk (i mod P)
  - Good load balance but always needs to read all the data
- Range-based partitioning on attribute A: tuple t goes to chunk i if $v_{i-1} < t.A < v_i$
  - Works well for range predicates but can suffer from data skew (data not evenly partitioned)
- Hash-based partitioning (see next slide)

Hash-based partitioning

- This way for distributing the tuples of a relation uses a hash function *h* that works on all values of the tuple (the function should really depend on all values, i.e., it should work in such a way that changing one component of the tuple t can change h(t) to any possible bucket number).
- For example, if we want P buckets for R, we might convert each component of the tuple t somehow to an integer between 0 and P-1, add the integers for all components, divide the result by P, and take the remainder as the bucket number for t. If P is also the number of processors, then we can associate each processor with a bucket and give that processor the content of the corresponding bucket.
- In general, hash-based partitioning provides good load balance but it works only for equality predicates and full scans

Parallel algorithm for selection

- We are interested in $\sigma_{A=v}(R)$ or $\sigma_{v1<A<v2}(R)$
- If the tuples of relation R are distributed evenly among the P processor's disks, then the elapsed time (time needed for completing the operation) on a parallel database with P processors is B(R)/P in all cases of data partitioning.
- However, different processors do the work:
  - Round robin: all servers do the work
  - Hash-based: if the hash function works on A, then one server for $\sigma_{A=v}(R)$, otherwise, all servers; all servers for $\sigma_{v1<A<v2}(R)$
  - Range-based: one server only

We do not count the cost of shipping the result to one site. In other words, we assume that we store tuple t in the result at the same processor that has t on its disk. Thus, the result will be in general stored distributed among the processors, just like R is. Note that the selection could radically change the distribution of tuples in the result compared to the distribution of R.

Parallel algorithm for projection

- Projection is similar to selection.
- If the tuples of relation R are distributed evenly among the P processor's disks, then the elapsed time on a parallel database with P processors is B(R)/P in all cases of data partitioning.
- The only difference with respect to selection is that projection cannot change the distribution of tuples.

Parallel algorithm for duplicate elimination

- Suppose we want to eliminate duplicates from relation R.
- If we used hash-based partitioning to distribute the tuples of R among processors (buckets), then duplicates tuples of R are placed at the same processor.
- If so, then we can produce the result in parallel by applying a standard algorithm at each processor for the corresponding portion of R. If we can do duplicate elimination in one pass at each processor, then the elapsed time is B(R)/P

Parallel algorithm for grouping

- Suppose we want to do grouping and aggregation on R
- If we used hash-based partitioning to distribute the tuples of R among processors (with a hash function that depends only of the grouping attributes), then tuples belonging to the same group of R are placed at the same processor.
- If so, each processor has all the tuples corresponding to one of the buckets of h, and then we can perform grouping and aggregation on these tuples locally, using any algorithms for grouping we have discussed earlier. If we can do grouping in one pass at each processor, then the elapsed time is B(R)/P

Parallel algorithm for binary operators

- Similarly to duplicate elimination, if the same hash function has been used to distribute the tuples of R and S, then we can take the union, intersection or difference of R and S by working in parallel on the portions of R and S at each processor.

- However, if different hash functions have been used to distribute the tuples of R and S, then we should first hash the tuples of R and S at each processor in parallel, using a single, new hash function (or, we can use one of the two hash functions, say the one used for R, also for S), and proceed according to the specific operator.

Parallel algorithm for set union

- If we have to distribute the tuples, then we use the hash function as in the first pass of the two-pass algorithm, but when the frame corresponding to a bucket i at processor j is full, instead of moving it to the disk at j, we ship the content of the frame to processor i (and, to save communication cost, if we have room for several pages per bucket in main memory, then we wait to fill several frames with tuples of bucket i before shipping them to processor i)
- Thus, processor i receives all the tuples of R and S belonging to bucket i. In the second stage, each processor performs the union of the tuples of R and S belonging to its bucket. As a result, the relation corresponding to the union will be distributed over all the processors. If the hash function h truly randomizes the placement of tuples in buckets, then we expect approximately the same number of tuples of the result to be at each processor.
- Set intersection, set difference, bag intersection, and bag difference can be performed just like union.

Parallel algorithm for join and grouping

- To compute the natural join between R(X,Y) and S(Y,Z), we do the same as before, but we use a hash function that depends only on the attributes Y, so that joining tuples are always sent to the same processor (bucket). We then perform the join at each processor using one of the algorithms we have discussed earlier.

Performance of parallel algorithms

- The total work (page accesses and CPU time) cannot be smaller for a parallel algorithm with respect to the uniprocessor version. However, because there are P processors working in parallel with P disks, we expect that the elapsed time is much smaller for the multiprocessor case that for the uniprocessor case.

- As we have seen, unary operations such as selection, and projection can be completed in a time which is 1/P of the time it would take to perform the operation at a single processor, provided relation R is distributed evenly.

Performance of parallel algorithms

- Consider the join operator, and assume that R and S are stored in one processor p. We remind the reader that we use a hash function on the join attributes that sends each tuple to one of the p buckets. To send the tuples of buckets i to processor i, for all i, we must read each tuple from disk in processor p, compute the hash function, and ship all tuples except the one out of P tuples that happens to belong to the bucket corresponding to p. Thus, we need to do B(R) + B(S) page accesses to read all the tuples of R and S and determine their bucket.
- We then must ship (B(R) + B(S)) × (P-1)/P pages across the network (only the (1/P)th of the tuples already in the right processor p need not be shipped).

Performance of parallel algorithms

In principle, we might suppose that the receiving processor has to store the data on its own disk and then execute the join on the tuples received.

If this is the case, and we can use the two-pass sort-merge join at each processor, the algorithm would use

- B(R) + B(S) page accesses for reading and hashing all tuples of R and S (we ignore here the cost of shipping such tuples).
- (B(R) + B(S))/P page accesses per processor, to account for the writing in the local disk during the distribution of tuples
- 3(B(R)+ B(S))/P page accesses at each processor for the local join (since the size of R will be B(R)/P and the size of S will be B(S)/P).

The total time is gone from 3(B(R) + B(S)) for uniprocessor to B(R) + B(S) + 4(B(R) + B(S))/P = (B(R) + B(S)) × (P+4)/P.

Performance of parallel algorithms

The cost could even decrease in certain situations.

- After storing the fragment at processor i, we could realize that we can use the one-pass algorithm for the local join; if this is true for all processors, we gain in execution time: B(R) + B(S) + 2(B(R) + B(S))/P.

- While receiving the buckets at processor i, we may realize that one of them (say that of S) is small enough to fit in the main memory of processor i. In this case, we can store and then retrieve only the larger bucket, say the R-bucket, whereas we simply keep in main memory the S-bucket while receiving it. The cost is B(R) + B(S) + 2B(R)/P.

- While receiving the buckets at processor i, we may realize that both buckets are small enough that together they fit in the main memory of processor i. In this case, we can work in main memory and the cost is B(R) + B(S).

Parallel algorithm for the cartesian product

Suppose that we want to compute the cartesian product of R and S. A possible way for a parallel implementation would be to broadcast the smallest relation to all the processors, and then partition the largest relation and compute a fragment of the cartesian product at each processor. The time to complete the operation is $B(R) \times B(S)/P$.

However, if we want to save broadcasting time, then we can do better: we organize the P processors in a $p_R \times p_S$ rectangle, such that $P = p_R \times p_S$, and then we identify each processor with a pair of coordinates. We then pick two hash functions, h_R (with range $1 \ldots p_R$) and h_S (with range $1 \ldots p_S$), and we send each tuple R(a) to all the processors with coordinates $(h_R(a),*)$, and we send S(b) to all the processors with coordinates $(*,h_S(b))$. After the data is partitioned. each processor locally computes the appropriate fragment of cartesian product.

Assuming that the hash functions behave well, each processor gets $B(R)/p_R + B(S)/p_S$ pages. To minimize this quantity, we have to make $B(R)/p_R = B(S)/p_S$ and so we can choose $p_R = \sqrt{P \times B(R)/B(S)}$ and $p_S = \sqrt{P \times B(S)/B(R)}$. Note that the elapsed time is again $B(R) \times B(S)/P$.

Parallel algorithm for multiway joins

Consider the query R(x,y), S(y,z), T(z,x), i.e., a natural join with three relations. The standard way to compute the result in parallel would be to proceed in two steps. In the first step we perform a parallel hash join between R and S to obtain an intermediate relation. In the second step we perform another parallel hash join between the intermediate relation and T. The potential problem is that the size of the intermediate relation could be high, which would mean expensive communication in the second step.

An alternative way for lowering communication cost is to organize the p processors in the 3-dimensional hypercube $P = p_x \times p_y \times p_z$ (the number of dimensions is the same as the number of variables in the arguments of the relations) . Each processor now identifies with a unique point in the 3-dimensional space. The algorithm uses a different hash function for each variable. The tuple R(a,b) will be sent to all processors with coordinates $(h_x(a), h_y(b),*)$. Similarly, each tuple S(b,c) will be sent to coordinates $(*, h_y(b), h_z(c))$ and each tuple T(c,a) to $(h_x(a),*,h_z(c))$. Each processor will compute the correct fragment in parallel. If we choose $p_x = p_y = p_z = p^{1/3}$, then each tuple will be replicated to $P^{1/3}$ processors.