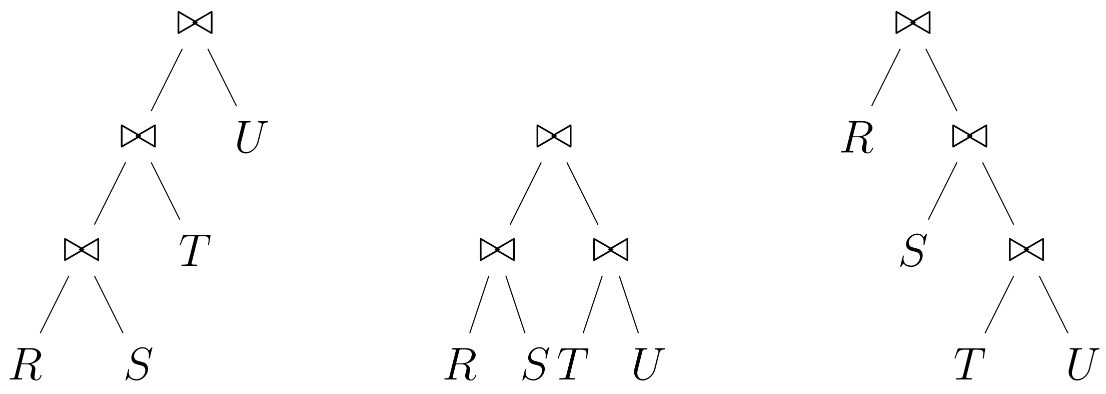
Join trees

For four relations, there are different structures for the join tree



Complexity:

- Decide on tree structure
- Assign relations to the leaves

For n relations, $n!$ assignments (and the number of tree structures is also very large)

Impractial to examine all possibilities; in practice only the structure on the left is sufficient

Another reason

Sometimes it is better to *pipeline* the results

- Whenever $R \bowtie S$ produces a block of data this is immediately joined with T, without writing to disk
- This cannot be done conveniently with the tree in the middle
- This is a reason to restrict to *left-deep* trees
- Another advantage: restrict the search space
- Why not right-deep? Because algorithms are usually designed for the former

Finding best right-deep plan

For up to three relations: No difference

For four relations: Omit the last three lines in the table

Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12.000
$((R \bowtie S) \bowtie U) \bowtie T$	55.000
$((T \bowtie U) \bowtie R) \bowtie S$	11.000
$((T \bowtie U) \bowtie S) \bowtie R$	3.000

Note that, for example

$$((S \bowtie T) \bowtie R) \bowtie U$$

could have been written

$$U \bowtie (R \bowtie (S \bowtie T))$$

but we intentionally chose to write it in a left-deep form

Physical query plans

- We study how to implement the algebraic operators, focussing on minimizing I/O
- "Iterator": a method to describe how operators for a physical plan can pass data between themselves
- Parameters: Besides B, T, V as before, we have $M + 1$, the size (in disk blocks) of main memory (using $M + 1$ instead of M makes the discussion more convenient)
- $M+1$ is really the number of blocks that are available to our algorithms (in practice this may not be fixed)

Scanning tables

- Trivial part of a plan: read all the contents of R
- *table scan*: Tuples are arranged in blocks (we always make this assumption) and we get the blocks one by one
- *index scan*: Using an index to get the blocks with the tuples
- *Sort scan*: Get the tuples in sorted order.
  - Using an index on the sort attributes
  - Sort in main memory (if the relation fits)
  - Use the 2-pass algorithm we describe later

  We return to table scan when we study selection

In our analysis, we assume that the data is stored on disk, but the result is left in main memory

- Why? Often we shall see that another operation can make direct use of the result
- But even if we do have to write the result to disk, this cost is fixed, and does not affect the *comparison* of different algorithms

Iterators: Implementation of physical operators in a way that the user gets the tuple one at a time. They consist of

- `Open()`. Initializes data structures, but does not get any tuples. Can range from trivial to implementing most of the operator
- `GetNext()`. Gets next tuple, and modifies data structures so that when called again, we get the following tuple. Returns `notFound` at end of relation
- `Close()`

Example: Table-scan

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
    }

GetNext{} {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
           RETURN NotFound;
        ELSE /* b is a new block */
           t := first tuple on block b;
     } /* now we are ready to return t and increment */
     oldt := t ;
     increment t to the next tuple of b;
     RETURN oldt;
}

Close() {
}
```

Sort-scan

- `Open()` does most of the work. It reads and sorts the relations, and sets a pointer to the first tuple
- `getNext()` just reads the next tuple and advances the pointer

Another example: Bag union.

- Iterators for both R and S, e.g., `R.Open()` and `S.Open()`
- Output tuples of R until we get to the end of the relation
- Then output tuples of S

```
Open() {
   R.Open();
   CurRel := R;
}

GetNext() {
   IF (CurRel = R) {
      t := R.GetNext() ;
      IF (t <> NotFound) /* R is not exhausted */
         RETURN t ;
      ELSE /* R is exhausted */ {
         S.Open() ;
         CurRel := S;
      }
   }
   /* here, we must read from S */
   RETURN S.GetNext();
   /* notice that if S is exhausted, S.GetNext()
      will return NotFound, which is the correct
      action for our GetNext as well */
}
Close() {
   R.Close() ;
   S.Close() ;
}
```

Classification of Algorithms

- One-pass algorithm. We each block of data at most once
- Two-pass algorithms. We may have to read (and write) each block twice

Which to use

- Depends on the size of the data and the type of operation
- For very large databases, it might be neccessary to use three or more pass algorithms, but this is rare
- We do not study three-path algorithms, but they are easy extensions of the two-pass ones

Trivial one-pass algorithms

Trivial one-pass algorithms: If the argument(s) take less than M blocks, read the two relations into memory and perform the operation in main memory

The one remaining block is used to store the result. Whenever it is full, we write part of the result to disk, erase the block and continue

From now on, we assume that the argument (unary) or at least one of the arguments (binary) takes more than M blocks

Note: For now, we postpone study of join

One-pass algorithms using only one block of main memory

Selection ($\sigma_C(R)$):

- Read one block of R into main memory
- Find the tuples that satisfy C
- Write them to the output
- Repeat until we have processed all of R

This does not depend on the size of R

This approach also works for projection and for bag union: read and write R a block at a time, then do the same for S

But this will not work for set union (why?)

One-pass algorithms for unary operations

Notation: Write m_1, . . . , m_{M+1} for the $M+1$ blocks of main memory

Recall that $B(R) > M$ so that we cannot read all of R

Can we evaluate $\delta(R)$?

- In general, no
- We need to keep at least one copy of each tuple to decide if the next tuple should be output or not
- But that might need more than M blocks

An alternative

- There are no non-trivial 1-pass algorithms for such operations
- Assume that we have a very reliable estimate for $B(\delta(R))$ (not $B(R)/2\ldots$) and we are sure that $B(\delta(R)) \leq M$.

Then we can use the following algorithm

1. Read one block of R into m_{M+1}
2. Write one copy of each different tuple into one of m_1, ..., m_M
3. Read a second block of R into m_{M+1}
4. Examine each tuple of this block. If it is not present in the m_1, ..., m_M blocks, write one copy of this tuple there
5. Once we have processed all of R, write the contents of m_1, ..., m_M

Does this work?

- As long as our estimate is correct, it does
- If not, we get overflow of main memory, we need to swap blocks out to disk, and the performance becomes very bad
- The same technique works for aggregation

We now turn to binary relations

- We assume that R is the smaller of the two relations
- Our algorithms work mostly for the case $B(R) \leq M$, $B(R)+B(S) > M$

Set Union

- Read all of R into m_1, ..., m_M and write them to the output (and construct an appropriate in-memory search structure)
- Read one block of S. For each tuple of s, s is in one of the m_is do nothing; otherwise write the tuple to the output
- Repeat until all of S has been read

Similar algorithms work for

- Set intersection
- Set difference (we must design separate algorithms for $R - S$ and $S - R$)
- Bag intersection and difference
- Product

Two-pass algorithms

Usually work when $B(R) \leq M^2$ (unary) or $B(R) + B(S) \leq M^2$ (binary)

They can be classified as based on

- Sorting
- Hashing
- Index

We start with sorting. The key part is a two-phase merge-sort

Offline merge-sort

Suppose we have M main-memory blocks, and M^2 blocks of data to sort

- Divide the data into M lists L_1, ..., L_M of M blocks each
- Read each list L_i into main memory, sort it, and write it out to disk in sorted blocks. We write $L_i = B_i^1, \ldots, B_i^M$, where B_i^1 contains the tuples that are first in this order
- Merge the sorted sublists. Initially, load the first block of each list into main memory

Note that we have not specified what attributes to sort by. This will depend on what we use the sorted lists for

- Find the first tuple among the blocks in main memory
- Move this tuple to the output block
- If the block from which first tuple was taken is now empty, read the next block (if there are any left) from the same sorted sublist into the same main memory block

Analysis

- Works as long as the number of lists is $\leq M$ and each list is no longer than M blocks, i.e., for data up to M^2 blocks
- Reading lists into main memory: $O(B(R))$ reads
- Writing sorted lists: $O(B(R))$ writes
- Reading the lists on the second pass: $O(B(R))$ reads
- Total I/O: $O(3B(R))$ (remember that we do not count the final write)

Duplicate elimination

Two-pass algorithm for $\delta(R)$ ($B(R) \leq M^2$)

- First pass: Sort into M lists in previous algorithm. Sort with respect to all attributes in an any order
- Read one block of each list into main memory. Let r be the first tuple under our order. It must be in main memory
- If there are any duplicates of r, they must be among the first tuples in each list, and so are in main memory. Find and delete them (if a main memory block becomes empty, read the next one from the list)
- Output one copy of r and repeat until we have processed all the data

I/O cost: $3O(B(R))$

Other operators

Similar techniques work for

- Aggregation
- Union (set-based)
- Intersection and difference

Two-Pass algorithms based on hashing

Example: $\delta(R)$

Hash-function h

$$h : t \rightarrow \{1, \ldots, M\}$$

where $t \in R$. The values are called *buckets*

Key properties of h

- $t_1 = t_2 \rightarrow h(t_1) = h(t_2)$. Trivial property, but means that duplicates are mapped to the same value
- The distribution of values of $h(t)$ for $t \in R$ is uniform. This means that the key to a successful implementation is the selection of h

A two-phase hash based function for $\delta(R)$ starts (first pass) by organizing the data into buckets, each of which corresponds to a list of blocks (one in main memory) of data with a specific hash-value

```
initialize M buckets using M empty blocks of memory;
FOR each block b of relation R DO BEGIN
   read block b into the (M+1)-th block of main memory;
    FOR each tuple t in b DO BEGIN
    IF the block for bucket h(t) has no room for t THEN
       BEGIN
          copy the block to disk;
          initialize a new empty block;
        END;
     copy t to the block for bucket h(t);
    END;
END;
FOR each bucket DO
    If the block for this bucket is not empty THEN
       write the block to disk;
```

How many blocks do we have for each bucket?

- If the hash-function is uniform, and
- $B(R) \leq M^2$,

then each bucket has at most M blocks

Furthermore, all duplicates are in the same bucket, so we can read each bucket into M blocks of main memory, and eliminate duplicates from each bucket

I/O cost is $3O(B(R))$

Similar techniques work for

- Aggregation
- Union
- Difference
- Intersection

Index-based algorithms

These are for selection (and, later, for join)

Selection $\sigma_{a=v}(R)$

- With a clustering index, the number of blocks is, on average, $\frac{B(R)}{V(R,a)}$
- We won't do such detailed anaylsis, but the number might be slightly higher, since
  - The index itself might be stored on disk
  - If the data needed k blocks it might actually take $k+1$, with the first and last being partially full
- Nonclustering index: number of blocks is around $\frac{T(R)}{V(R,a)}$
- Note that this could be greater than $B(R)$. In that case it makes sense not to use the index, and use a table-scan instead

Example

$B(R) = 1.000$, $T(R) = 20.000$. Suppose an index on a.

- Without using the index, I/O cost is 1.000
- $V(R, a) = 100$, clustering index. Index-based algorithm has cost $\frac{1.000}{100} = 10$
- $V(R, a) = 10$, nonclustering index. Index-based algorithm has cost $\frac{20.000}{10} = 2.000$, higher than table-scan
- $V(R, a) = 20.000$, i.e., a is a key. Index-based algorithm (clustering or nonclustering index) has cost 1

- Queries such as $\sigma_{a \geq 10}$ can also be implemented efficiently with an index
- Complex conditions: $\sigma_{a=v \texttt{ AND } C}$ can be implemented by using an index for a and then filtering with respect to C.
- A condition such as $a = v$ AND $b = w$ can be implemented by using either an index on a or one on b (if both exist), so we should check which is more efficient