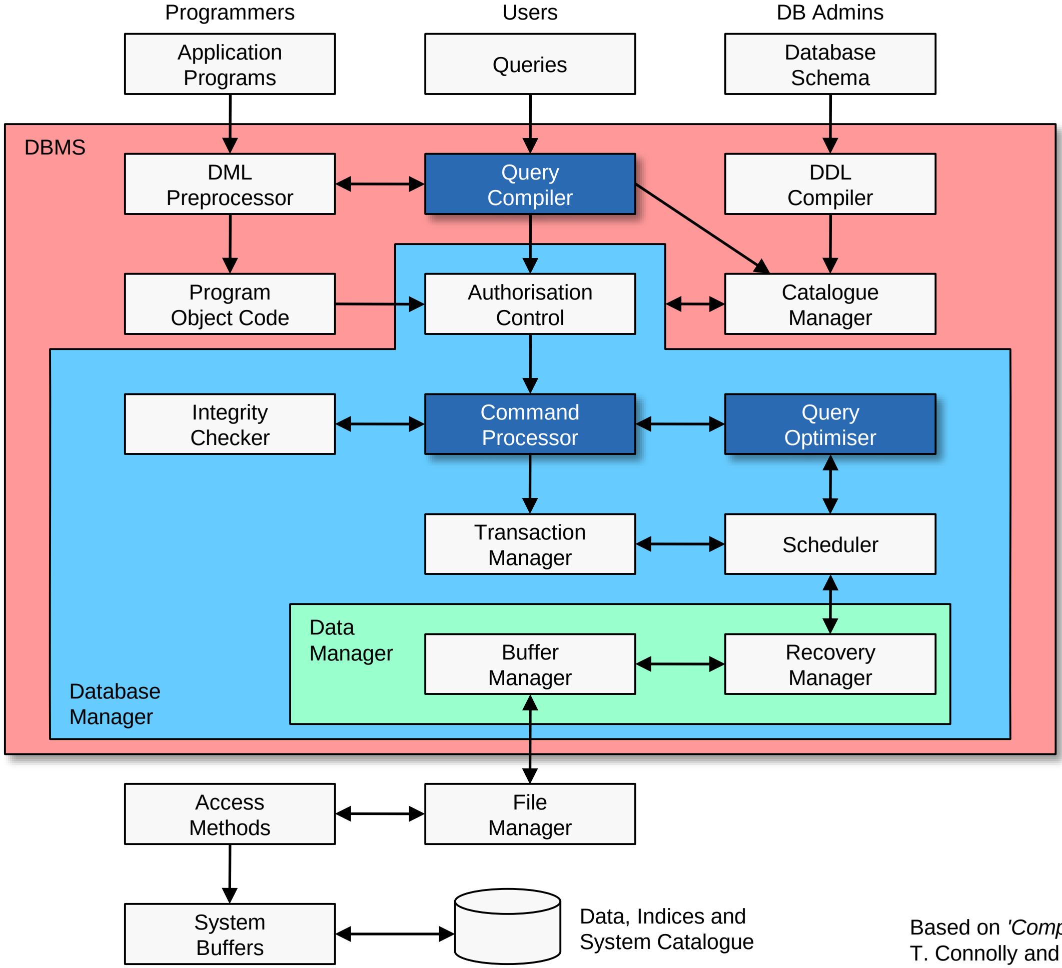
Introduction to Databases
Query Processing and Optimisation

Prof. Beat Signer

Department of Computer Science
Vrije Universiteit Brussel

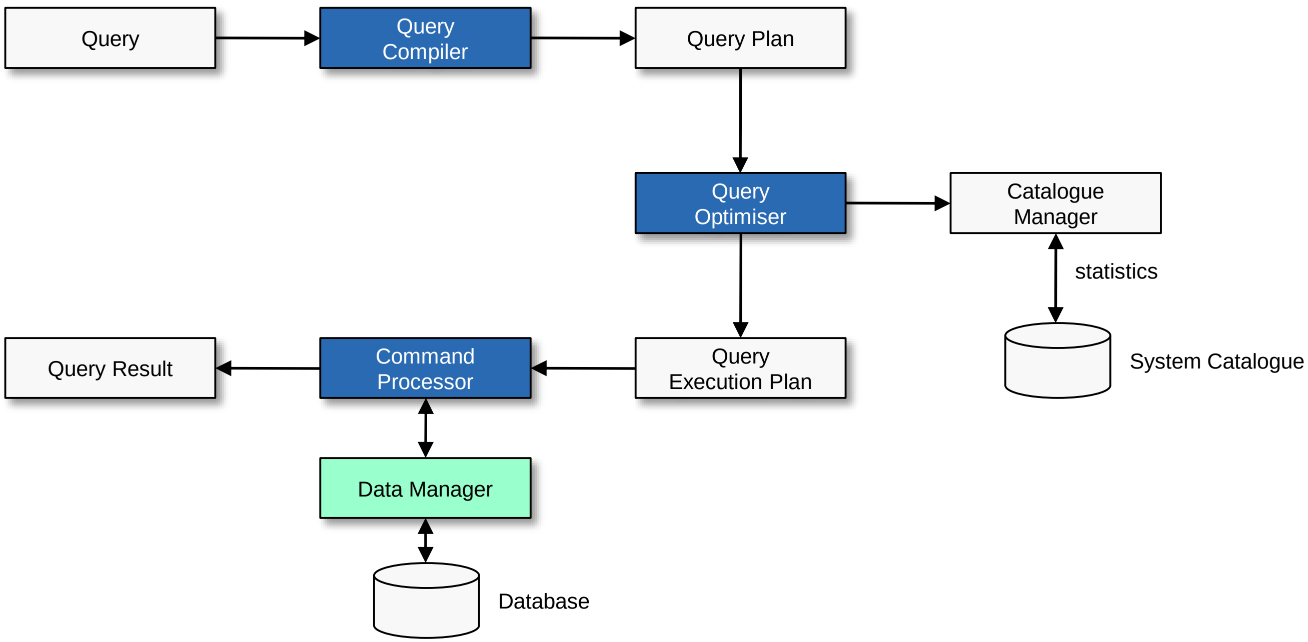
http://www.beatsigner.com

Context of Today's Lecture



Based on *'Components of a DBMS'*, Database Systems, T. Connolly and C. Begg, Addison-Wesley 2010

Basic Query Processing Steps



Basic Query Processing Steps ...

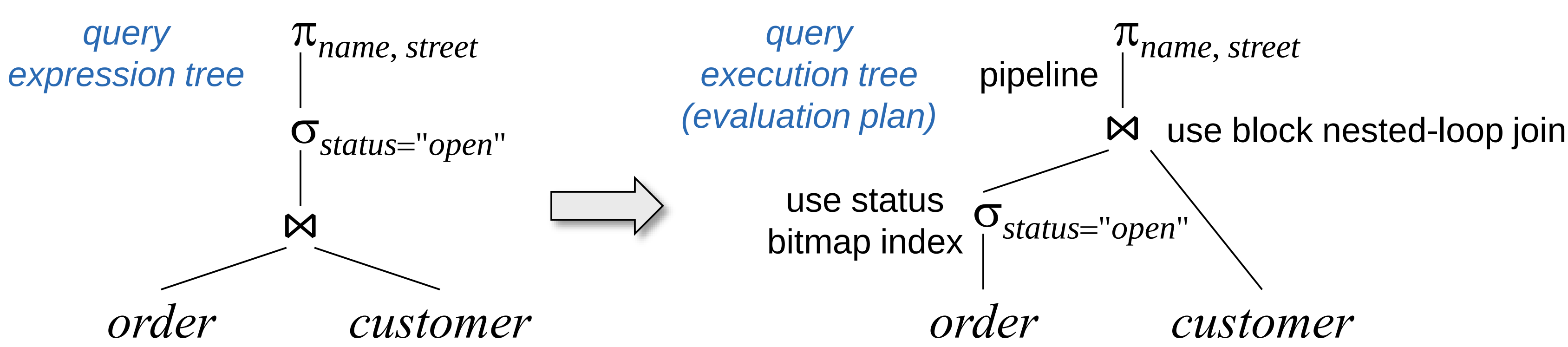
- Query parsing and translation (*query compiler*)
  - check the syntax (e.g. SQL for relational DBMS)
  - verify that the mentioned relations do exist and replace views
  - transform the SQL query to a *query plan* represented by a relational algebra expression (for relational DBMS)
    - different possible relational algebra expressions for a single query
- Query optimisation (*query optimiser*)
  - transform the initial query plan into the best possible query plan based on the given data set
    - specify the execution of single query plan operations (*evaluation primitives*)
      - e.g. which algorithms and indices to be used
    - the *query execution plan* is defined by a sequence of evaluation primitives
- Query evaluation (*command processor*)
  - execute the query execution plan and return the result

Query Expression and Execution

```sql
SELECT name, street
FROM Customer, Order
WHERE Order.customerID = Customer.customerID AND status = 'open';
```

- Transform the SQL query to the following query plan

$$\pi_{name,\ street}(\sigma_{status="open"}(order \bowtie customer))$$



note that we will later see how to optimise the query expression tree

Query Costs

- The *query costs* are defined by the *time to answer a query* (process the query execution plan)
- Different factors contribute to the query costs
  - disk access time, CPU time or even network communication time
- The costs are often dominated by the *disk access time*
  - *seek time* (t_S) (~4 ms)
  - *transfer time* (t_T) (e.g. 0.1 ms per disk block)
    - write operations are normally slower than read operations
- For simplicity, we will use the *number of block transfers* and the *number of seeks* as cost measure
  - real systems may also take CPU costs into account

Query Costs ...

- We often compute the *worst case costs* where the main memory buffer can hold only a few blocks
  - we further assume that data has to be initially read from disk and is not yet in the buffer from a previous operation

Selection Operation

- The lowest-level query processing operator for accessing data is the *file scan*
  - search and retrieve records for a given *selection condition*
- In the following we discuss different file scan algorithms
  - we assume that blocks of the file are stored continously
- *Linear search*
  - given a file with n blocks, we scan each block and check if any records satisfy the condition
  - a selection on a candidate key attribute (unique) can be terminated after a record has been found
    - average costs: $t_S + n/2 * t_T$, worst case costs: $t_S + n * t_T$
  - applicable to any file regardless of ordering, the availability of indices or the type of selection operation

Selection Operation ...

- *Binary search*
  - an equality selection condition on a file that is ordered on the selection attribute (n blocks) can be realised via a binary search
  - *note that this only works if we assume that the blocks of the file are stored continously!*
  - worst case costs: $\lceil \log_2(n) \rceil * (t_S + t_T)$

Index-based Selection Operation

- A search algorithm that makes use of an index is called an *index scan* and the index structure is called *access path*
- *Primary index* and *equality* on *candidate key*
  - retrieve a single record based on the index
  - costs for a B+-tree with height h: $(h+1) * (t_S + t_T)$
- *Primary index* and *equality* on *non-candidate key*
  - multiple records might fulfil the condition (possibly spread over n successive blocks)
  - costs for a B+-tree with height h: $h * (t_S + t_T) + t_S + n * t_T$
- *Secondary index* and *equality* on *candidate key*
  - retrieve a single record based on the index
  - costs for a B+-tree with height h: $(h+1) * (t_S + t_T)$

Index-based Selection Operation ...

- *Secondary index* and *equality* on *non-candidate key*
  - each matching record may be in a different block (matching records spread over n blocks)
  - costs for a B⁺-tree with height h: $(h + n) * (t_S + t_T)$
  - for large number of blocks n with matching records, this can be very expensive and cost even more than a linear scan!
- *Primary index* and *comparison* on attribute A
  - we assume that the relation is sorted on attribute A
  - $\sigma_{A \geq v}(r)$
    - use index to find the first record that has a value of $A \geq v$ and do a *sequential file scan* from there
  - $\sigma_{A \leq v}(r)$
    - sequential file scan until $A \geq v$ without using any index

Index-based Selection Operation ...

- *Secondary index* and *comparison* on attribute A
  - $\sigma_{A \geq v}(r)$ or $\sigma_{A \leq v}(r)$
  - for a B$^+$-tree index we can scan the leaf index blocks from the smallest value to v or from v to the largest value
  - each record may be in a different block (spread over n blocks)
  - for large number of records n, this can be very expensive and cost even more than a linear scan!

Conjunctive Selection Operation

- A conjunctive selection has the form $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$
- *Conjunctive selection* using a *single index*
  - check if there is an access path available for an attribute in one of the simple conditions θ_i
  - use one of the approaches described before (with minimal cost) to retrieve the records and check the other conditions in memory
- *Conjunctive selection* using a *composite index*
  - use the appropriate multi-key index if available
- *Conjunctive selection* using *multiple indices*
  - requires indices with record pointers
  - retrieve record pointers from different indices and perform an intersection of the sets of record pointers
    - additional conditions (without index) might be checked in memory

Disjunctive Selection Operation

- A disjunctive selection has the form $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$
- *Disjunctive selection* using *indices*
  - indices can only be used if there is an index for *all* conditions; otherwise a linear scan of the relation has to be performed anyway

Sorting

- Sorting in database systems is important for two reasons
  - a query may specify that the *output* should be *sorted*
  - the *processing* of some relational query operations can be implemented *more efficiently* based on sorted relations
    - e.g. join operation
- For relations that fit into memory, techniques like quicksort can be used
- For relations that do not fit into memory an *external merge sort* algorithm can be used

External Merge Sort Example

11 blocks

initial relation

o	8
e	2
x	0
m	5
l	9
o	7
e	17
t	12
r	25
a	3
s	19
r	8
t	13
n	29
i	11
b	20
x	3
d	12
f	21
f	3
w	4
g	5

create runs

runs (4 blocks each)

R_0

e	2
e	17
l	9
m	5
o	8
o	7
t	12
x	0

R_1

a	3
b	20
i	11
n	29
r	25
r	8
s	19
t	13

R_2

d	12
f	21
f	3
g	5
w	4
x	3

merge

sorted output

a	3
b	20
d	12
e	2
e	17
f	21
f	3
g	5
i	11
l	9
m	5
n	29
o	8
o	7
r	25
r	8
s	19
t	12
t	13
w	4
x	0
x	3

assumption in this example: memory can hold at most $M = 4$ blocks

External Merge Sort

- Let us assume that there is space for M memory blocks

(1) Create runs

- repeatedly read M blocks of the initial relation, sort them and write them back as run R_i (resulting in a total of N runs)

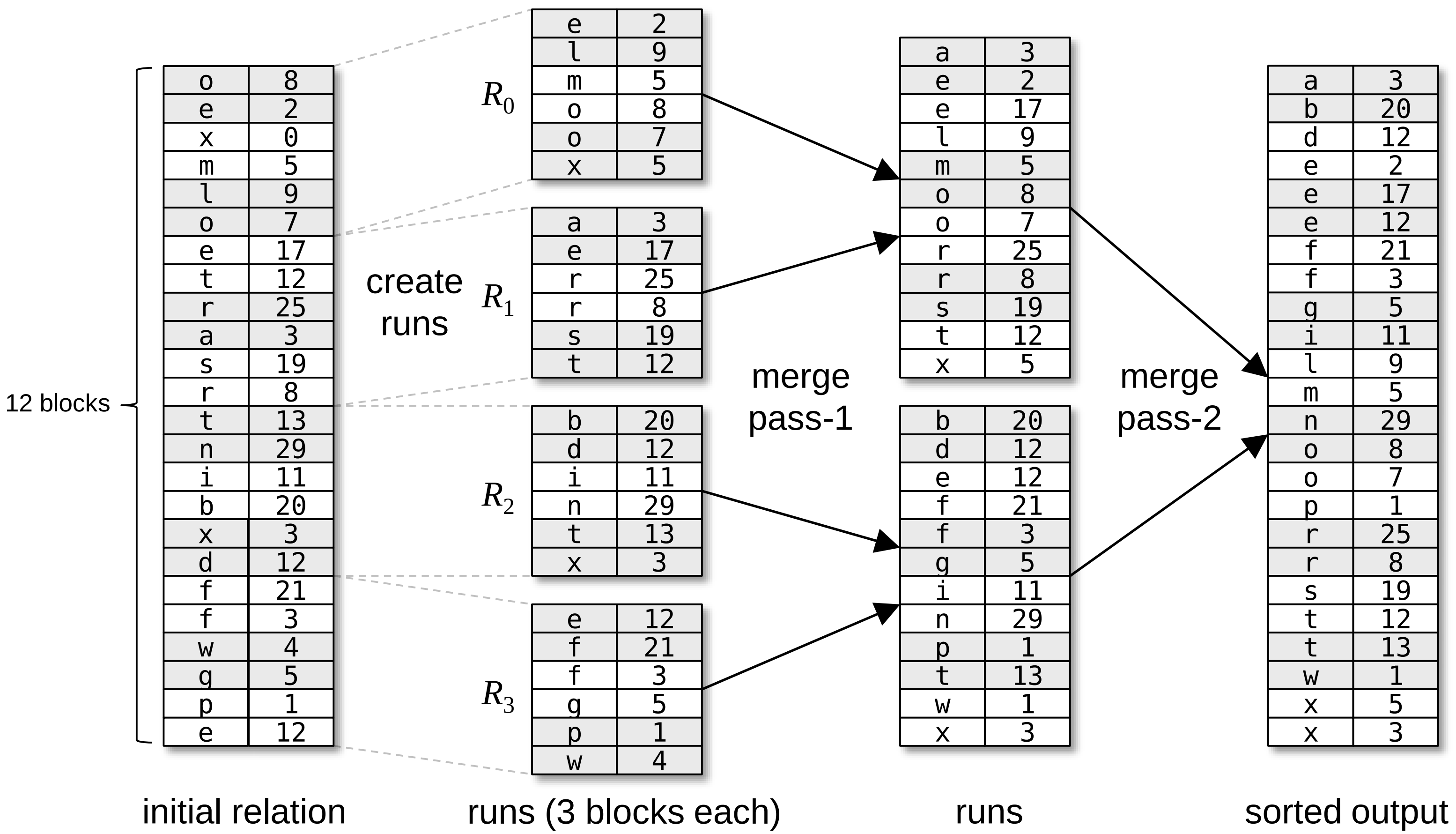
(2) Merge the runs (*N-way merge*), for $N < M$

- use N memory blocks to buffer the input runs (one block per run) and one block as an output buffer
- repeat the following steps until all input buffer blocks are empty
  - select the smallest record r_s from all input runs and write it to the output block
    - if the output block is full then write it to the disk
  - remove the record r_s from the buffered block of run R_i
    - if the buffered block of run R_i becomes empty, then fetch the next block of the input run R_i into the buffer

External Merge Sort ...

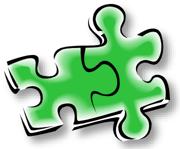
- If $N \geq M$ then *multiple merge passes* are required
  - in each pass continuous groups of $M - 1$ runs are merged
  - *each pass reduces the number of runs by a factor $M - 1$*
- Cost analysis
  - initial number of runs: $\lceil B/M \rceil$
  - for a file with B blocks we need $\lceil \log_{M-1}(B/M) \rceil$ merge passes
  - creation of the initial runs requires a read and write of each block
    - $2B$ block transfers
  - each pass reads every block and writes it to the disk
    - $2B$ block transfers per run
    - last run forms an exception since the blocks do not have to be written to disk
  - the total number of block transfers for an external merge sort is therefore $B * (2 * \lceil \log_{M-1}(B/M) \rceil + 1)$

External Merge Sort Example



assumption in this example: memory can hold at most $M = 3$ blocks

Join Operation



- Different algorithms for implementing join operations
  - nested-loop join
  - block nested-loop join
  - index nested-loop join
  - merge join
  - hash join
- The query optimiser may choose an algorithm based on cost estimates
- In the join algorithm examples, we will use the two relations `Customer` and `Order` with the following data
  - `Customer` has 5000 records and 100 blocks
  - `Order` has 10 000 records and 300 blocks

Nested-Loop Join

```
for each tuple tr in r {
  for each tuple ts in s {
    if (tr and ts satisfy the join condition θ) {
      add tuple tr × ts to the result set
    }
  }
}
```

- A nested-loop join with the *outer relation* r and the *inner relation* s can be used to compute a theta join $r \bowtie_\theta s$
- The nested-loop join algorithm requires no indices and can be used for any join condition
- A nested-loop join is *expensive* since every pair of tuples in the two relations has to be examined

Nested-Loop Join ...

- Let us assume that r has b_r blocks and n_r tuples and s has b_s blocks and n_s tuples
- In the *worst case*, the buffer can only hold one block of each relation r and s
  - $n_r * b_s + b_r$ block transfers and $n_r + b_r$ seeks
    - e.g. `Customer` in outer relation: $5000 * 300 + 100 = 1\,500\,100$ block transfers and $5000 + 100 = 5100$ seeks
    - e.g. `Order` in outer relation: $10\,000 * 100 + 300 = 1\,000\,300$ block transfers and $10\,000 + 300 = 10\,300$ seeks
- In the *best case*, both relations fit into memory
  - $b_r + b_s$ block transfers and 2 seeks
- If at least one relation fits into memory, that relation should be made the inner relation

Block Nested-Loop Join

```
for each block Br of r {
  for each Block Bs of s {
    for each tuple tr in Br {
      for each tuple ts in Bs {
        if (tr and ts satisfy the join condition θ) {
          add tuple tr × ts to the result set
        }
      }
    }
  }
}
```

- Variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation

Block Nested-Loop Join ...

- Much better worst case performance than nested-loop join
  - $b_r * b_s + b_r$ block transfers and $2 * b_r$ seeks
    - e.g. `Customer` in outer relation: $100 * 300 + 100 = 30\,100$ block transfers and 200 seeks
- Other optimisations
  - if the join attributes in a natural join or an equi-join form a candidate key on the inner relation, the inner loop can terminate on the first match
  - scan inner loop *alternately forward and backward*
    - buffered data from previous scan can be reused

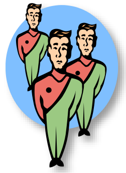
Indexed Nested-Loop Join

- In an *indexed nested-loop join* we use an index on the inner loop's join attribute for equi-joins/natural joins
  - index lookups instead of file scans
  - for each tuple t_r of the outer relation r, the index is used to lookup tuples in the inner relation s
  - index might even be constructed just to compute the join
- Worst case performance
  - buffer has space for one block of the outer relation r and we need an index lookup on s for each tuple in r
    - cost: $b_r * (t_S + t_T) + n_r * c$, where c is the cost for a single selection on s
  - e.g. 30-ary B$^+$-tree index on `Order` relation
    - tree height not greater than $\lceil \log_{15}(10\,000) \rceil = 4$
    - cost: $100 * (t_S + t_T) + 5000 * (4+1)\,(t_S + t_T) = 25\,100 * (t_S + t_T)$

Other Join Implementations

- *Merge join*
  - sort both relations on the join attribute
  - merge the sorted relations to join them
- *Hybrid merge join*
  - one relation is sorted and there exists a secondary B+-tree index on the join attribute for the second relation
    - *merge* the sorted relation with the *leaf address* entries of the B+-tree
    - *sort* the result set *on the addresses of the unsorted relation's tuples*
    - scan the unsorted relation to fetch the data and replace the pointers
- *Hash join*
  - uses a hash function to partition the tuples of the relations r and s based on the join attributes
  - details about hash and merge join can be found in the book

Duplicate Elimination



- Duplicates can be eliminated via sorting or hashing
  - when sorting, *duplicates will be placed next to each other* and all but one instance of the duplicate tuples can be removed
  - duplicates can be eliminated in the different steps of an external merge sort
    - while the runs are generated
    - during the intermediate merge passes
  - hashing places duplicate tuples into the same bucket
- The elimination of duplicates has high costs and therefore SQL does not eliminate duplicates by default
  - has to be explicitly specified via the `DISTINCT` keyword
- A *projection* can be implemented by performing the projection on each tuple and eliminating duplicates

Set Operations

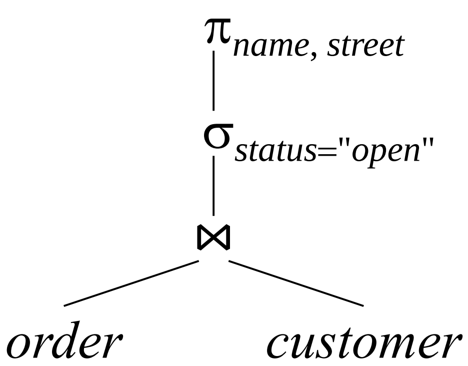
- The *union* (∪), *intersection* (∩) and set *difference* (–) operators can be implemented based on a variant of *merge join* after sorting or a variant of the *hash join*
- Hash implementation
  - partition the relations r and s by using a single hash function h which results in the hash buckets H_{r_i} and H_{s_i}
  - $r \cup s$
    - build an in-memory index of H_{r_i} and add the tuples of H_{s_i} not yet present
    - add the tuples in the hash index to the result
  - $r \cap s$
    - build an in-memory index of H_{r_i} and for each tuple in H_{r_i} probe the index H_{s_i} and add the tuple to the result only if it is present in the hash index
  - similar implementation for difference $r - s$
    - remove tuple from H_{r_i} if present in H_{s_i}

Expressions

- The individual relational operations that have been discussed so far normally form part of more *complex expressions*
- There are two approaches how a query execution tree can be evaluated
  - *materialisation*
    - compute the result of an evaluation primitive and materialise (store) the new relation on the disk
  - *pipelining*
    - pass on tuples to parent operations even while an operation is still being executed

Materialisation

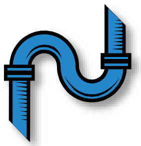
- Evaluate one operation after another starting at the leave nodes of the query expression tree
  - materialise intermediate results in temporary relations and use those for evaluating operations at the next level



1. compute *order* $\bowtie$ *customer* and store relation
2. compute $\sigma_{status="open"}$ on materialised relation and store
3. compute $\pi_{name,\ street}$ on materialised relation

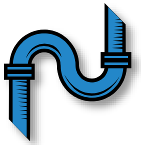
- A materialised evaluation is always possible
  - costs of reading and writing temporary relations can be quite high
  - *double buffering* with two output buffers for each operation

Pipelining



- *Pipelining* evaluates *multiple operations simultaneously* by passing results of one operation to the next one *without storing the tuples on the disk*
- Much cheaper than materialisation since no I/O operations for temporary relations
- Pipelining is not always possible
  - e.g. does not work for input for sorting algorithms
- Pipelines can be executed in a *demand driven* or in a *producer driven* manner
- Demand driven or *lazy pipelining* (*pull pipelining*)
  - top level operation repeatedly requests the next tuple from its children

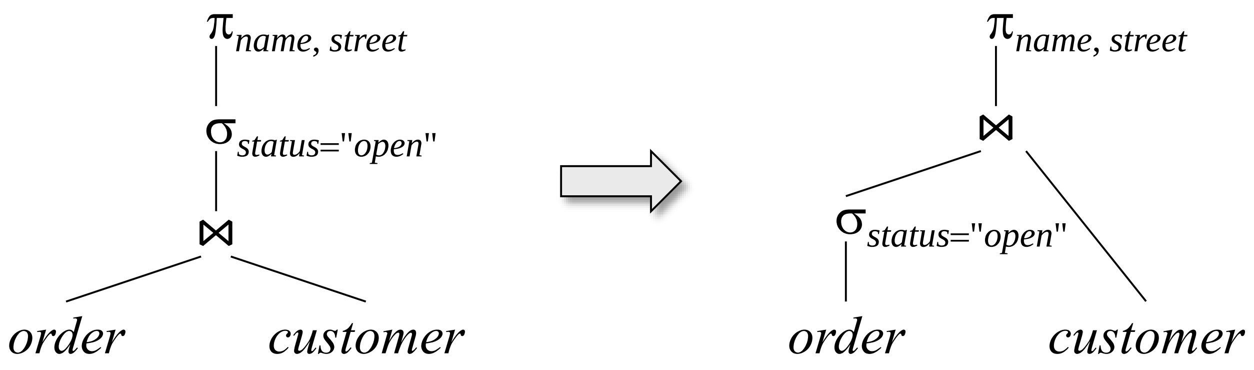
Pipelining ...



- Producer driven or *eager pipelining* (*push pipelining*)
  - the child operators produce tuples eagerly and pass them to their parents via a buffer
  - if the buffer is full, the child operator has to wait until the parent operator consumed some tuples
- The use of pipelining may have an *impact on the types of algorithms that can be used* for a specific operation
  - e.g. join with a pipelined left-hand-side input
    - the left relation is never available all at once for processing
    - i.e. merge join cannot be used if the inputs are not sorted
    - however, we can for example use an indexed nested-loop join

Query Optimisation

- There are alternative ways for evaluating a given query
  - different equivalent expressions (query expression trees)
  - different potential algorithms for each operation of the expression



Query Optimisation

- There can be enormous differences in terms of performance between different query evaluation plans for the same query
  - e.g. seconds vs. days to execute the same query
- Cost-based query optimisation
  (1) generate logically equivalent expressions by using a set of *equivalence rules*
  (2) annotate the expressions to get alternative query evaluation plans (e.g. which algorithms to be used)
  (3) select the cheapest plan based on the estimated costs
- Estimation of query evaluation costs based on
  - statistical information from the catalogue manager in combination with the expected performance of the algorithms

Equivalence Rules

- Conjunctive selection operations can be deconstructed into a sequence of individual selections
  - $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- Selection operations are *commutative*
  - $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- Cascade of projection operations (only final one)
  - $\pi_{A_1}(\pi_{A_2}(\ldots(\pi_{A_n}(E))\ldots)) = \pi_{A_1}(E)$
- Selections can be combined with cartesian products and theta joins
  - $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
  - $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules ...

- Theta join (and natural join) operations are *commutative*
  - $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
  - note that the order of attributes is ignored
- Natural join operations are *associative*
  - $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- Theta joins are *associative* in the following manner
  - $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
  - where θ_2 contains attributes only from E_2 and E_3
- Union and intersection operations are *commutative*
  - $E_1 \cup E_2 = E_2 \cup E_1$
  - $E_1 \cap E_2 = E_2 \cap E_1$

Equivalence Rules ...

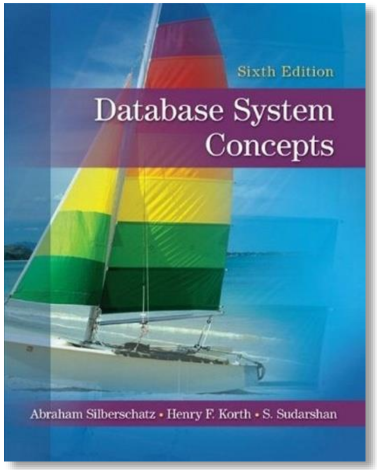
- Union and intersection operations are *associative*
  - $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
  - $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
- The *selection* operation *distributes* over *union*, *intersection* and *set difference*
  - $\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$
- The *projection distributes* over the *union* operation
  - $\pi_{\mathrm{A}}(E_1 \cup E_2) = (\pi_{\mathrm{A}}(E_1)) \cup (\pi_{\mathrm{A}}(E_1))$
- Note that this is only a selection of equivalence rules

Heuristic Optimisation

- Cost-based optimisation can be expensive
  - a DBMS may use some *heuristics* to reduce the number of cost-based choices
- A heuristic optimisation transforms the query expression tree by using a set of rules that typically improve the execution performance
  - perform *selection as early as possible*
    - reduces the number of tuples
  - perform *projection as early as possible*
    - reduces the number of attributes
  - perform *most restrictive selection and join operations* (smallest result size) before other operations

Homework

- Study the following chapters of the *Database System Concepts* book
  - chapter 12
    - sections 12.1-12.8
    - Query Processing
  - chapter 13
    - sections 13.1-13.7
    - Query Optimization



Exercise 10

- Query Processing and Query Optimisation

References

- A. Silberschatz, H. Korth and S. Sudarshan, *Database System Concepts* (Sixth Edition), McGraw-Hill, 2010

Vrije Universiteit Brussel

Next Lecture

Transaction Management