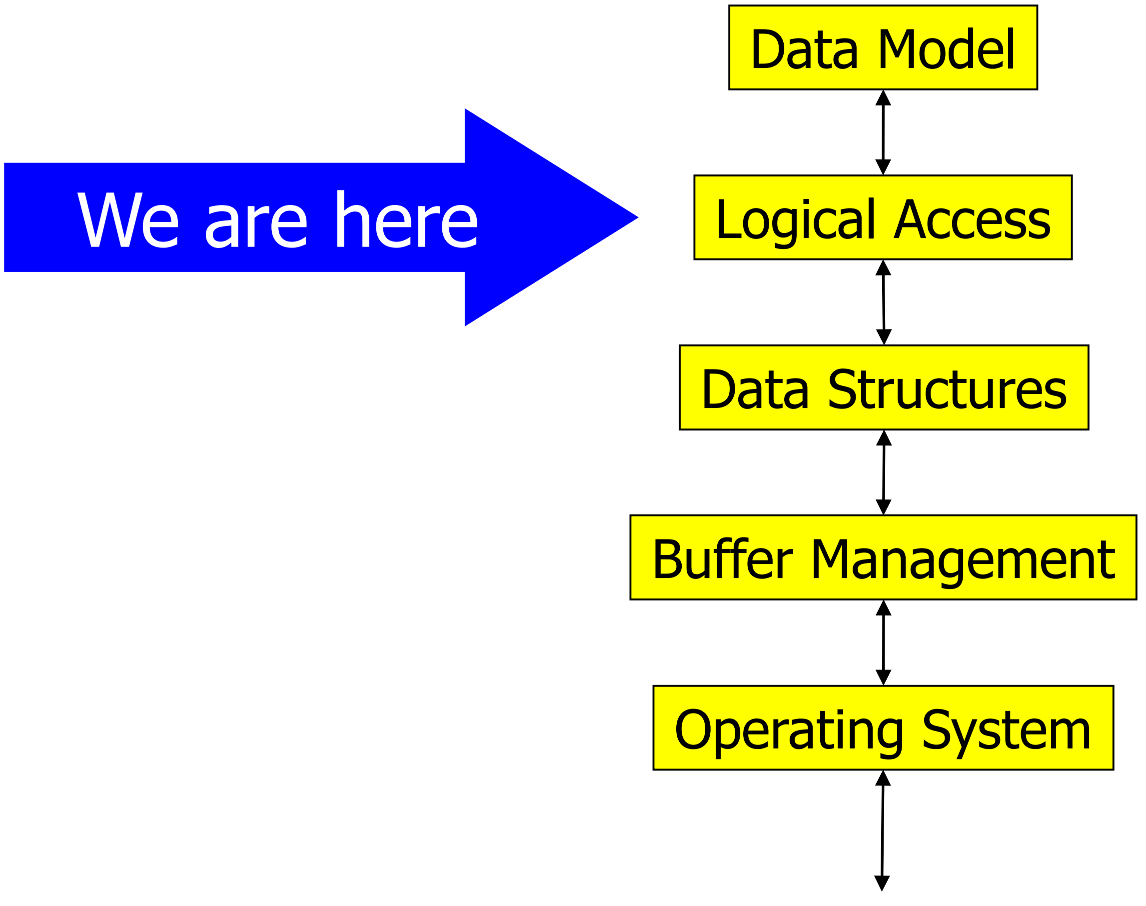
Datenbanksysteme II: Query Execution

Ulf Leser

Content of this Lecture

- Overview: Query optimization
- Relational operators
- Query execution models
- Implementing (some) relational operators

5 Layer Architecture



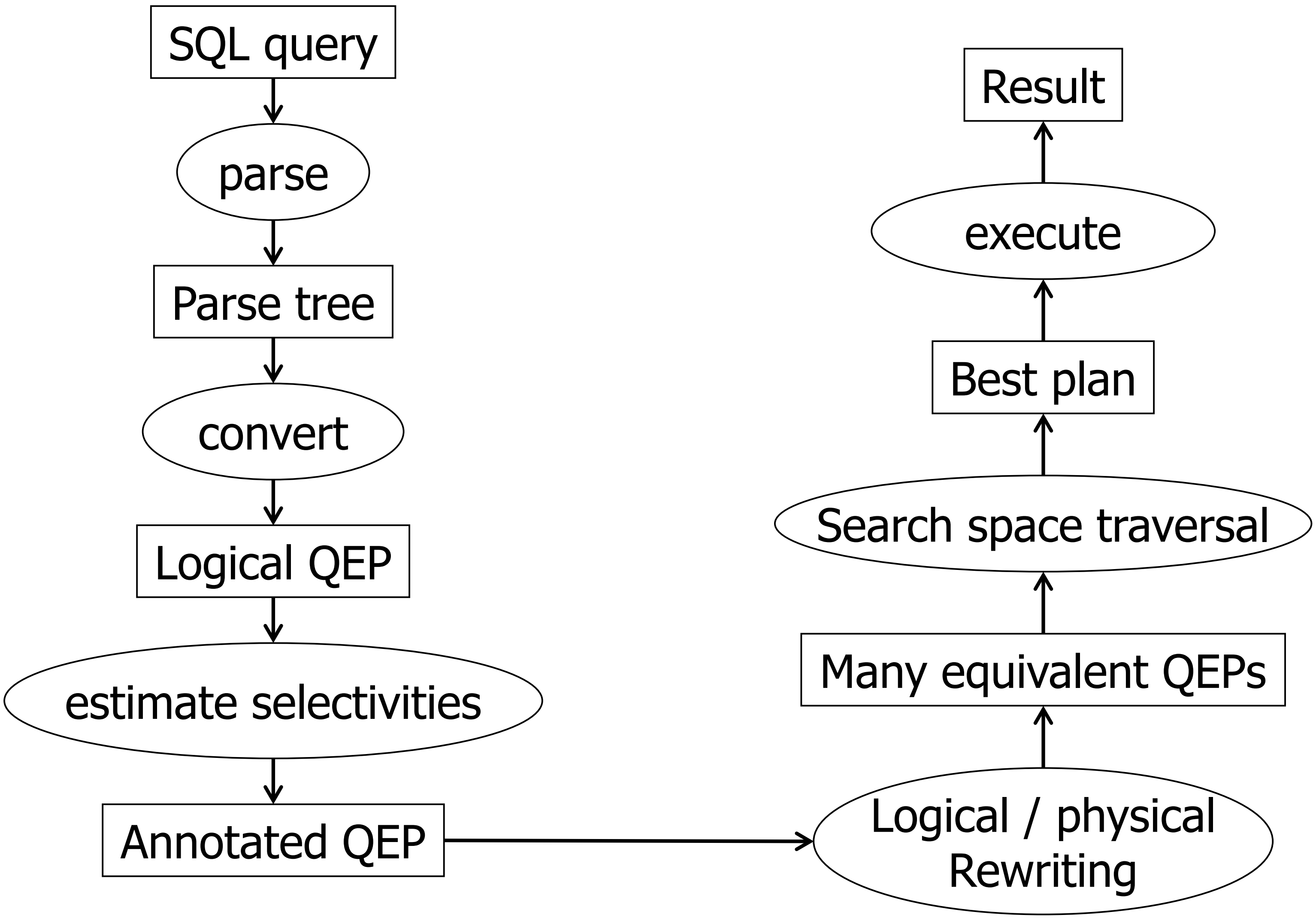
Query Optimization

- We have
  - Structured Query Language SQL
  - Relational algebra
  - How to access tuples in many ways (scan, index, …)
- Now
  - Given a SQL query
  - Find a fast way and order of accessing tuples from different tables such that the answer to the query is computed
  - Usually, we won't find the best way, but avoid the worst
  - Use knowledge about value distributions, access paths, query operators, IO cost, …
  - Compile a declarative query in an "optimal" executable program

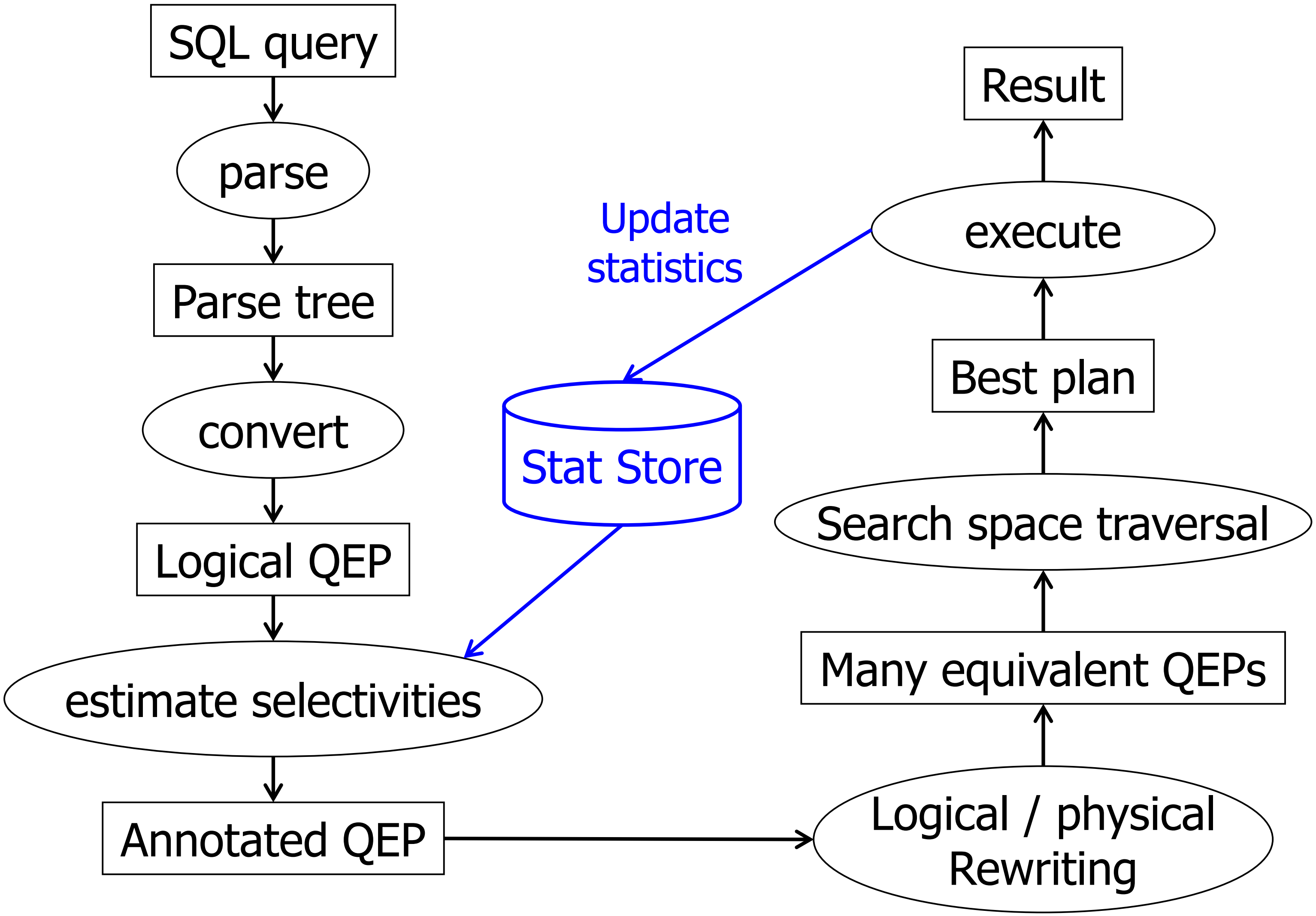
Steps (Sketch)

- Translate query in a logical query execution plan (QEP)
  - Structured representation of a relational algebra expression
- Logical optimization: QEPs are rewritten in other, semantically equivalent and hopefully faster QEPs
  - E.g., selection is commutative: $\sigma_A(\sigma_B(expr)) = \sigma_B(\sigma_A(expr))$
- Physical optimization: For each (relational) operator in the query, we have multiple possible implementations
  - Table access: scan, indexes, sorted access through index, …
  - Joins: Nested loop, sort-merge, hash, …
- Query execution: Execute the best query plan found

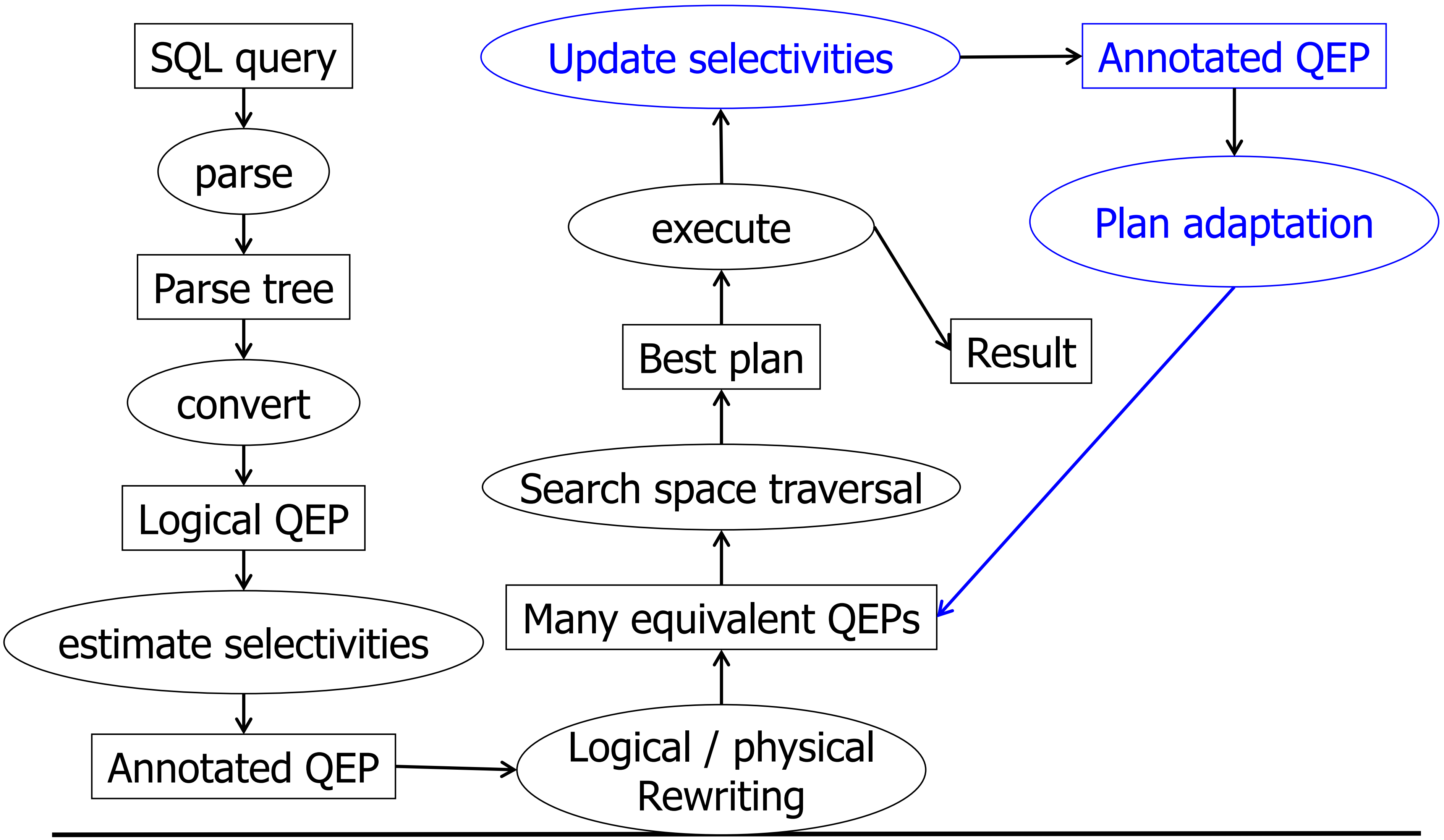
Overview Optimization



Overview Optimization



Adaptive Optimization



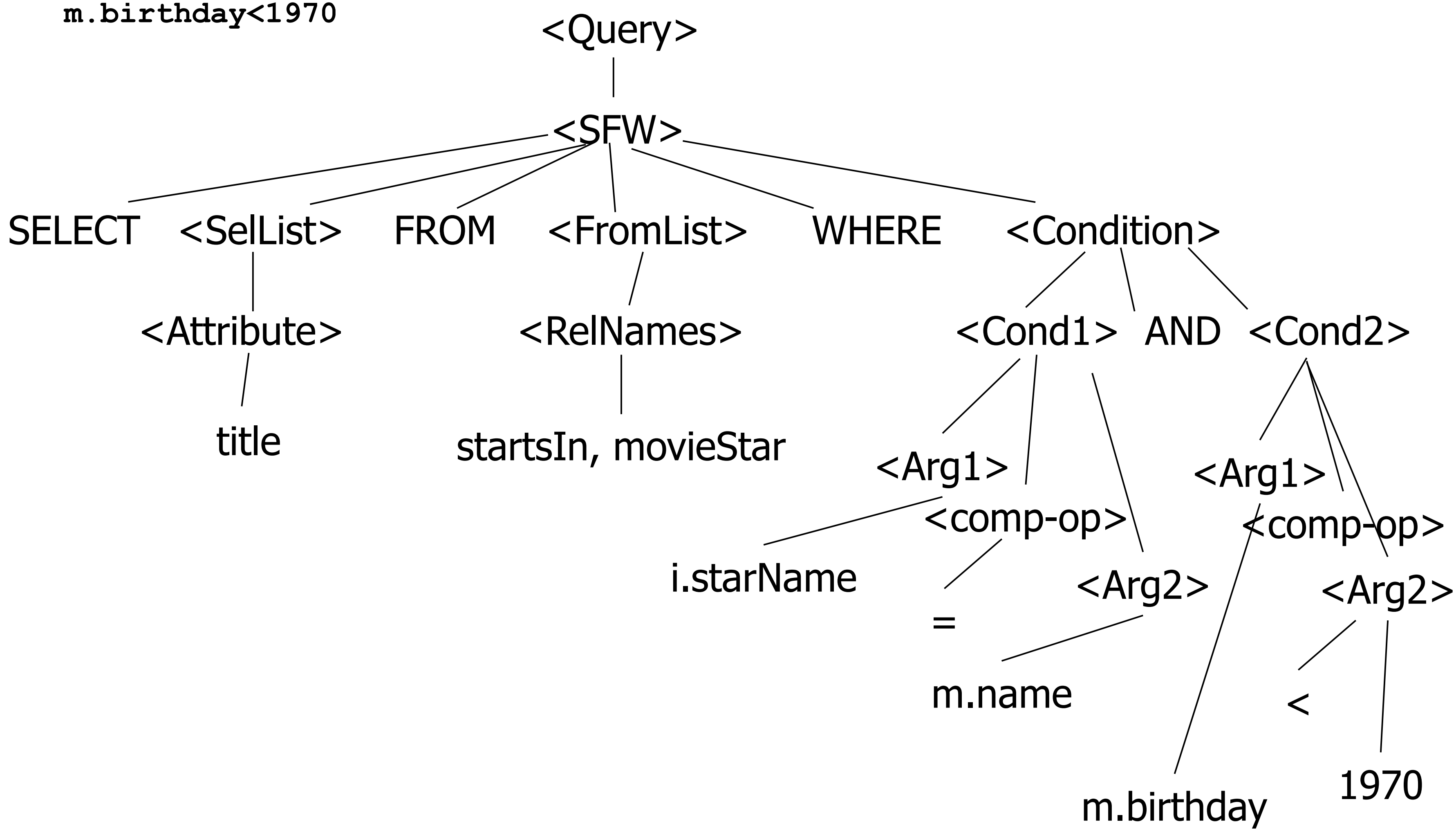
Example SQL query

```
SELECT title
FROM   starsIn i, movieStar m
WHERE  i.starName = m.name AND
       m.birthday<1970;
```

(Find all movies with stars born before 1970)

```
SELECT title
FROM   starsIn i, movieStar m
WHERE  i.starName = m.name AND
       m.birthday<1970
```

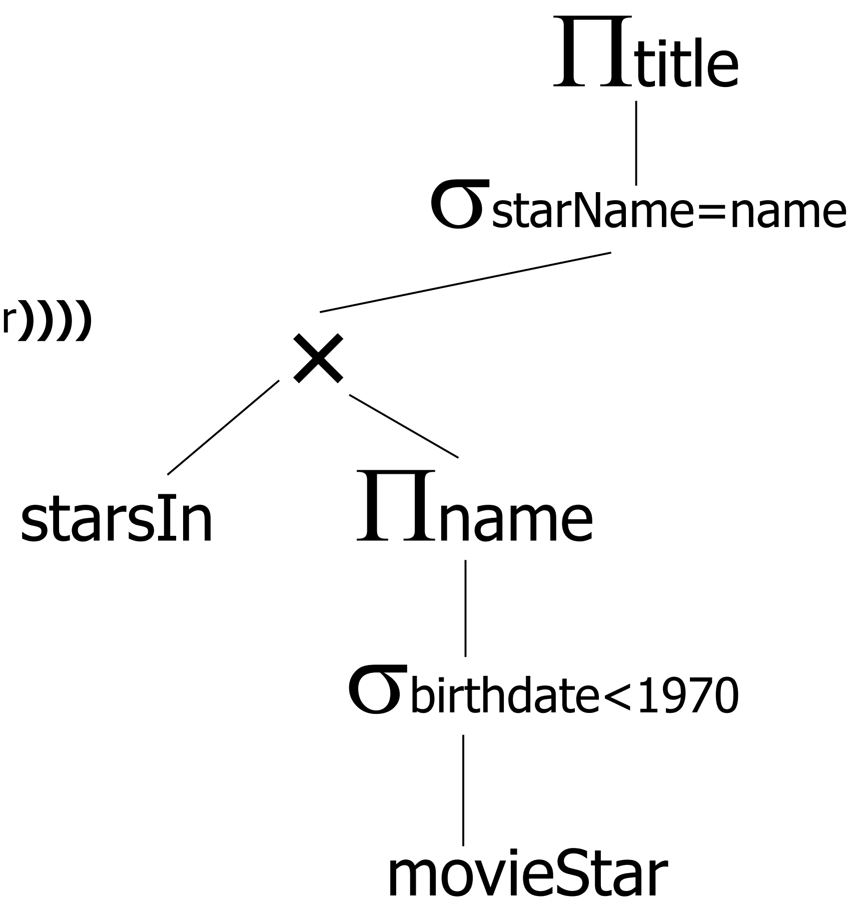
Parse Tree



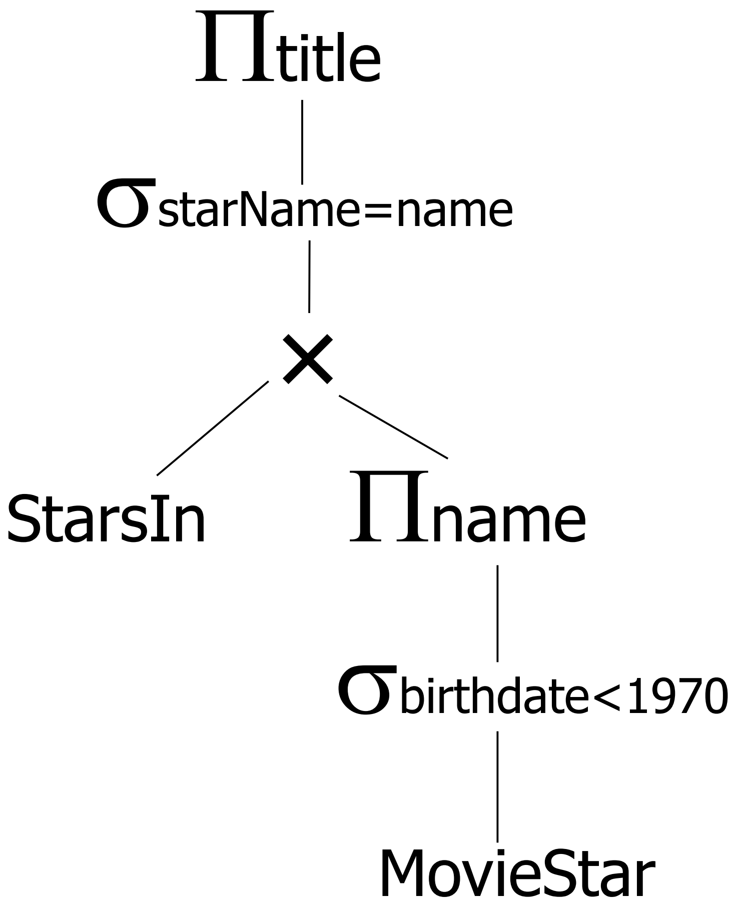
Relational Algebra / Logical Query Plan

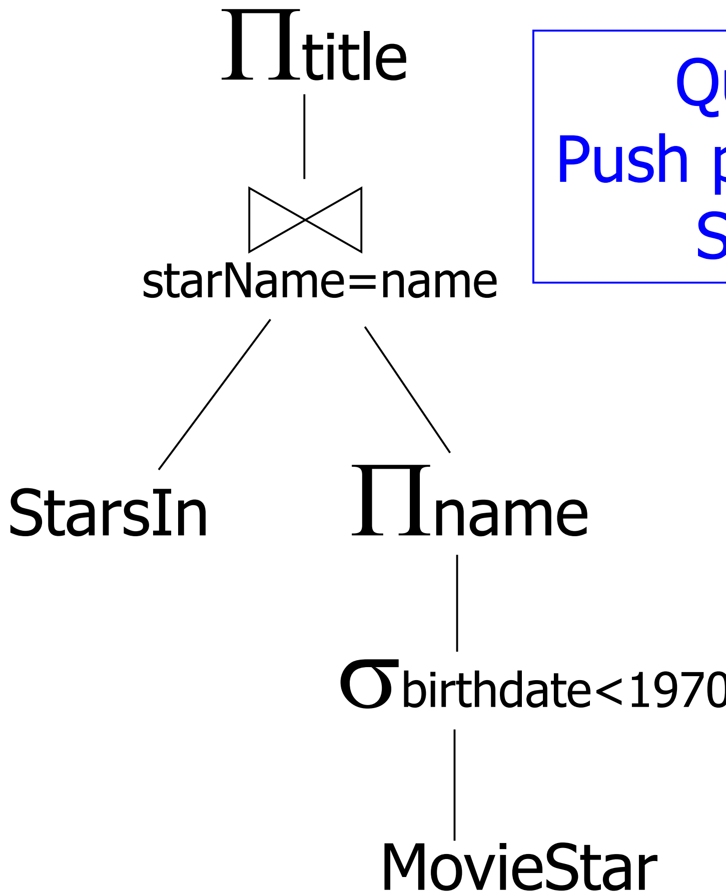
```
SELECT title
FROM   starsIn i, movieStar m
WHERE  i.starName = m.name AND
       m.birthday<1970
```

$$\Pi_{title}\left(\sigma_{starName=name}\left(starsIn \times \Pi_{name}\left(\sigma_{birthdate<1970}(movieStar)\right)\right)\right)$$



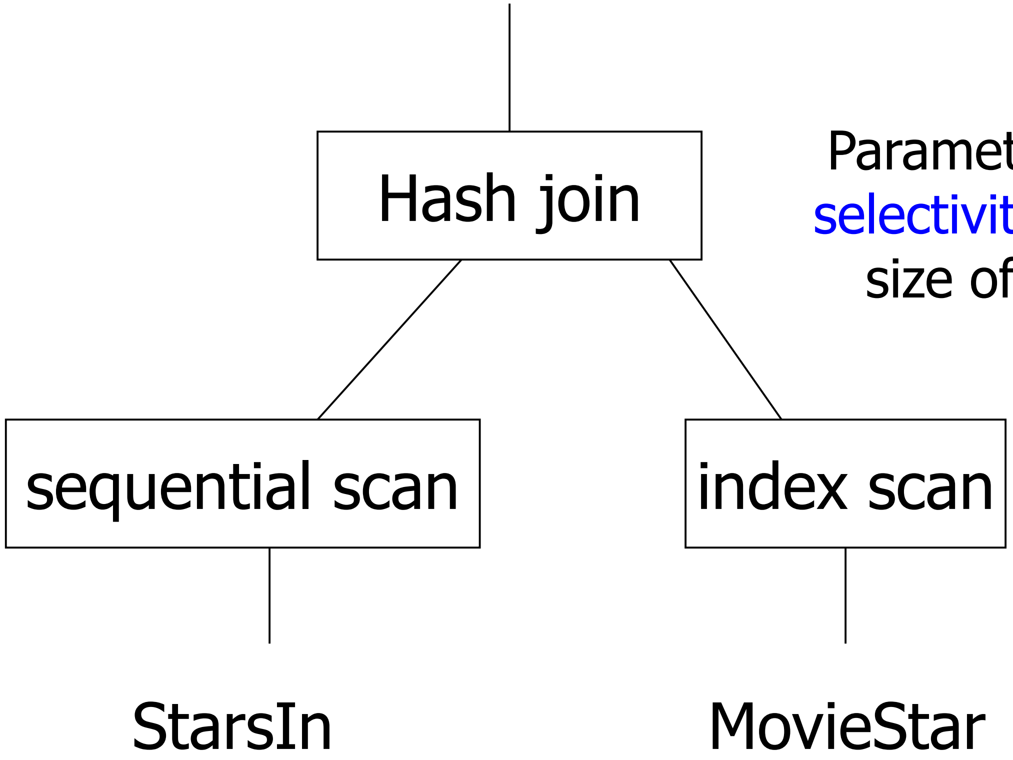
Improved Logical Query Plan





Question:
Push projection to StarsIn?

Physical Plan



Parameters: Join order, selectivity, memory size, size of attributes, …

Parameters: Selectivity, fragmentation of data file, size of tuples, ,...

Overview

- Today: Implementation of one-table relational operators
  - Projection, selection, scans, group-by
- Next topic: Physical join implementations
  - Blocked nested loop, sort-based, hash-based
- Next: Query optimization
  - Query rewriting, plan reordering
- Next: Cost estimation
  - For cost-based query optimization

Content of this Lecture

- Overview: Query optimization
- Relational operators
- Query execution models
- Implementing (some) relational operators

Relational Operations: One Table

- In the following: Table means table or intermediate result
- Selection σ: WHERE clause
  - Read table and filter tuples based on condition
  - Selection never increases table length (selectivity)
  - Conjunctions, disjunction, equality, negation, …
  - [A join is a selection, but special treatment in order]
  - Implementation: Scan or index (depending on selectivity)
- Projection π: SELECT clause
  - Read tuples and manipulate columns
  - With DISTINCT clause: Duplicates must be filtered
  - Projection usually decreases breadth of table – smaller result size
    - When not?
  - Implementation: While computing results

One Table cont'd

- Group-by: Grouping and aggregation
  - Put all tuples with equal values in all grouping attributes into one bag; output one tuple per bag by aggregating other values
  - Reduces number of tuples (how much?)
  - Implementation by sorting or hashing
- Distinct: Duplicate elimination
  - Read table and remove all duplicate tuples
  - Implementation by sorting or hashing
- Order-by: Sorting
  - Always last clause in query, but injected often by optimizer
  - Pipeline breaker
  - Implementation: In-memory or external sorting

Relational Operations: Two Tables

- Cartesian product x
  - Read two tables and build all pairs of tuples
  - Usually avoided – combine product and selection to join
  - Products in a plan are hints to wrong queries
  - Specified implicitly by FROM clause
  - Implementation: No tricks (if really requested)
- Join $\bowtie$
  - All pairs of tuples matching the join condition
  - Natural join, theta join, equi join, semi join, outer join
  - Expensive, often very selective – favorite target of optimizers
  - Possibility: Join-order and join implementation
  - Specified implicitly or explicitly in WHERE clause
  - Implementation: Sort-based, blocked-nested loop, hash, zigzag, …

Relational Operations: Two Queries

- Union $\cup$
  - Read two tables and build union (by identity) of all tuples
  - Duplicates are removed (alternative: UNION-ALL)
  - Requires tables to have same schema
- Intersection $\cap$
  - Read two tables and build intersection (by identity) of tuples
  - Requires tables to have same schema
  - Same as join over all attributes
- Minus $/$
  - Subtract tuples of one table from tuples from the other
  - Requires tables to have same schema

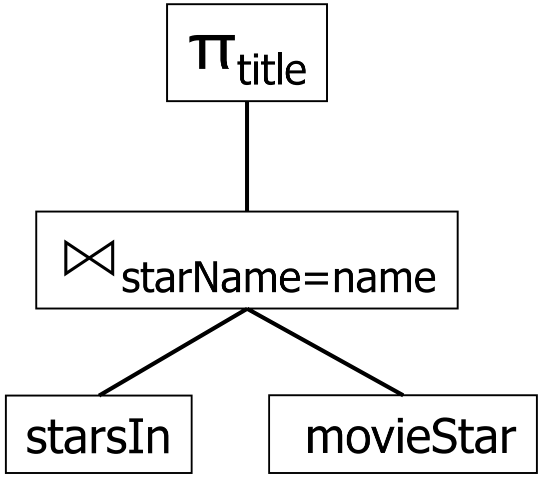
Content of this Lecture

- Overview: Query optimization
- Relational operators
- Query execution models
- Implementing (some) relational operators

Query Execution

- Typical model: Operator implementations call each other to pass tuples up the tree
  - Iterator concept: Open, next, close
    - Each operator implementation needs these three methods
  - Produces deep stacks and many push/pops
  - Plan generation is simple: Nesting of operations
- Two modes: Blocked, Pipelined
  - Blocked: Most work done in open
  - Pipelined: Most work done in next
  - Pipeline-breaker only allow blocked mode (e.g. sorts)
- Modern alternative: Compile into function-free program

Example – Blocked (Sketch)




```
p = projection.open();
while p.next(t)
    output t;
p.close();
```

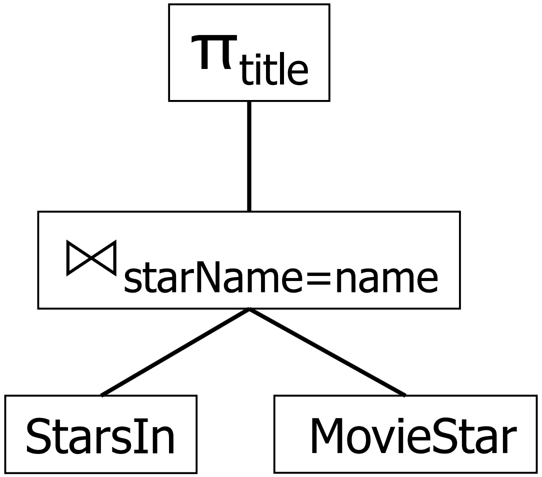
```
class projection {
open() {
  j = join.open();
  while j.next(t)
    tmp[i++]=t.title;
  j.close();
  cnt:=0;
}
next(t) {
  if (cnt<tmp.max)
    t = tmp[cnt++];
    return true;
  else return false;
}
close() {
  discard(tmp);
}
}
```

```
class join {
open() {
  l = table.open(starsIn);
  while l.next(tl)
    r = table.open(movieStar)
    while r.next(tr)
      if tl.starname=tr.name
        tmp[i++]=tl⋈tr;
    r.close();
  end while;
  l.close();
  cnt:=0;
}
next(t) {
 if (cnt<tmp.max)
    t = tmp[cnt++];
    return true;
  else return false;
}
close() {
  discard( tmp);
}}
```

Blocked Execution

- Traditional model
- Advantages
  - Does always work, for all operators
  - Simple to understand and implement
  - Highly extensible (common open-fetch-close API)
  - Classical optimization goal: Minimize size of intermediate results
- Disadvantages
  - Requires large buffers in memory
  - Leads to "blocked" result arrival – difficult for downstream apps
    - Think of web side display
  - Difficult to parallelize (no operator parallelism)

Example – Pipelined (Sketch)



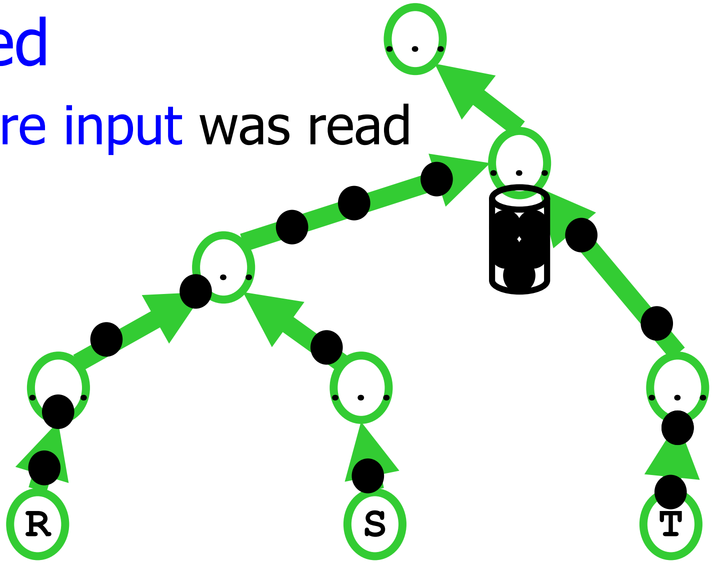

```
p = projection.open();
while p.next(t)
    output t;
p.close();


class projection {
open() {
  j = join.open();
}
next(t) {
  if j.next( t)
    t = title;
    return true;
  else
    return false;
}
close() {
  j.close();
}
}
```

```
class join {
open() {
  l = table.open(starsIn);
  r = table.open(movieStar);
  l.next( tl);
}
next( t) {
  if r.next(tr)
    if tl.starname=tr.name
      t=tl⋈tr;
      return true;
    else
      next (t);
  else
    if l.next(tl)
      r.close();
      r = table.open(movieStar);
      return next(t);
    else
      return false;
}
close() {
  l.close();
  r.close();
}}
```

Pipelined versus Blocked

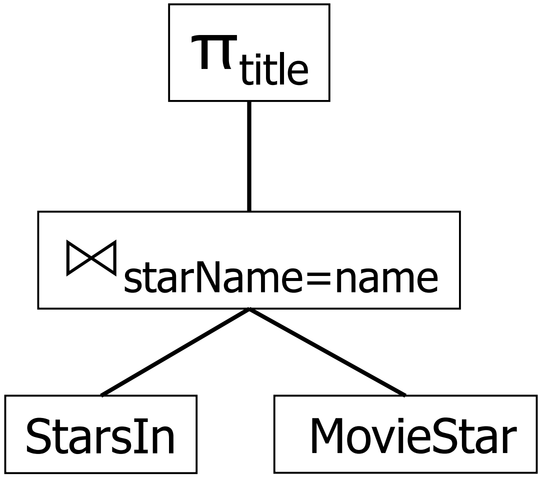
- Pipelining much preferred
  - Very little demand for buffer space
    - When intermediate results are large, buffers need to be stored on disk
  - Different ops within query can be assigned to different threads
    - Overlapping execution
  - Results come early and continuously
- Pipeline breaker cannot be pipelined
  - next() can be executed only after entire input was read
  - Examples
    - Sorting
      - Exception: When input is sorted
    - Grouping and aggregation
      - Depending on implementation
    - Minus, intersection



Non Binary: Pipelined versus Blocked

- Projection with duplicate elimination
  - When implemented with sorting – pipeline breaker
  - But: Recall implementation without sorting
  - next() can return early
  - But we need to keep track of all values already returned – requires large buffer

Example – Compiled (Sketch)




```
l = table.open(starsIn);
r = table.open(movieStar);
go = l.next(tl);
while go do
  while r.next(tr)
    if tl.starname=tr.name
      t=tl⋈tr;
      output t.title;
  end while;
  if l.next( tl)
    r.close();
    r = table.open(movieStar);
  else
    l.close();
    r.close();
    go = false;
end while;
```

Content of this Lecture

- Overview: Query optimization
- Relational operators
- Query execution models
- Implementing (some) relational operators

Select versus Update

- We do not discuss update, delete, insert
- Update and delete usually have embedded queries – "normal" optimization
  - But: data tuples must be loaded (and locked and changed and persistently written if TX not rolled-back)
  - Some tricks don't work any more
- Insert may have query (INSERT …. AS SELECT …)

Implementing Operators

- Most single table operations are straight-forward
  - See book by Garcia-Molina, Ullmann, Widom for detailed discussion
- We sketch three single table operations
  - Scanning a table
  - Duplicate elimination
  - Group By
- Joins are more complicated – later

Scanning a Table

- At the bottom of each operator tree are relations
- Accessing them implies a table scan
  - If table T has b blocks, this costs b IO
- Often better: Combine with next operation in plan
  - `SELECT t.A, t.B FROM t WHERE A=5`
  - Selection: If index on t.A available, perform index scan
    - Assume |t|=n, |A|=a different values, z=n/a tuples
      - Index has height ~$\log_k(n)$
      - Scan B+ index and find all matching TIDs
      - Accessing z tuples from t costs 1 to z IO (sequential or random)
    - Especially effective if A is a key: Only one tuple selected, 1 IO on table
  - Projection: Integrate into table scan
    - Read complete tuples, but only pass-on attributes that are needed
      - Why not read partial tuples?

Scanning a Table 2

- Conditions can be complex

```
SELECT t.A, t.B FROM t
WHERE A=5 AND (B<4 OR B>9) AND C='müller' …
```

- Approach
  - Compute conjunctive normal form
  - Independent indexes: Find TID lists for each conjunct, then intersect
  - With MDIS: Directly find matching TIDs
  - Without indexes: Scan table and evaluate condition for each tuple
- For complex conditions and small tables, linear scanning usually is faster
  - Depends on expected result size
  - Cost-based optimization required

Duplicate Elimination

- Option 1: Sorting
- Sort table on DISTINCT columns
  - Can be skipped if table is already sorted
- Scan sorted table and output only unique tuples
- Generates output in sorted order (for later reuse)
- Pipeline breaker (see later)

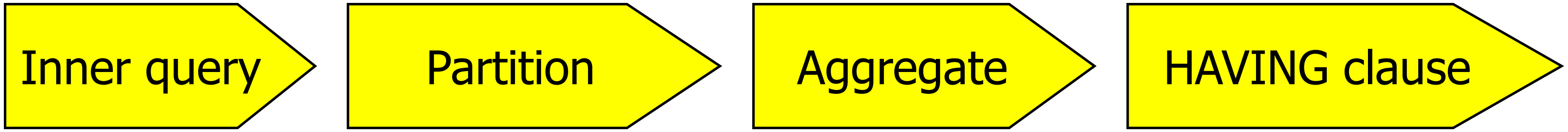
Duplicate Elimination

- Option 2: Use hashing
- Scan table and build hash table H on all unique values
  - Needs good hash function, avoid conflicts
- When reading a tuple, check if it has already been seen
  - If not: insert tuple and copy it to the output; else: skip tuple
  - No pipeline breaker
  - Does not sort result (but existing sorting would remain)
- No pipeline breaker
- Memory: Problem; assumes H to fit in memory

Grouping and Aggregation

```
SELECT day_id, sum(amount*price)
FROM   sales S
GROUP  BY day_id
```

- Recall: SELECT may contain only GROUP BY attributes and aggregate functions
- Partition result of “inner query” by GROUP BY attributes
- For each partition, compute one result tuple: GROUP BY attributes and aggregate function applied on values of other attributes in this partition
  - Note: Depending on the aggregate function, we might need to buffer more than one value per partition – examples?



Implementing GROUP BY

- Proceed like duplicate elimination
- Also keep to-be-aggregated attributes
  - Raw (e.g. median), intermediate (e.g. sum/count), aggregated (count, sum)
- Eventually, compute the aggregated columns
  - Simple: SUM, COUNT, MIN, MAX, ANY
  - More memory required: AVG, Top-5, median
- Pipelining? Same properties as for duplicate elimination

Computing Median

- Option 1: Partition table into k partitions
  - Scan table
  - Build (hash) table for first k different GROUP BY values
  - When reading one of first k, add value to (sorted) list
  - When reading other GROUP value, discard
  - When scan finished, output median of k groups
  - Iterate – next k groups
  - Can adapt (k) to number of groups, assumes groups of similar sizes
- Option 2: Sort table on GROUP BY and Median attribute
  - Then scan sorted data
  - Buffer all values per group
  - When next group is reached, output middle value
- What if we cannot buffer all values of a group?