# Datenbanksysteme II: Implementation of Database Systems

## Query Execution

Material von

Prof. Johann Christoph Freytag

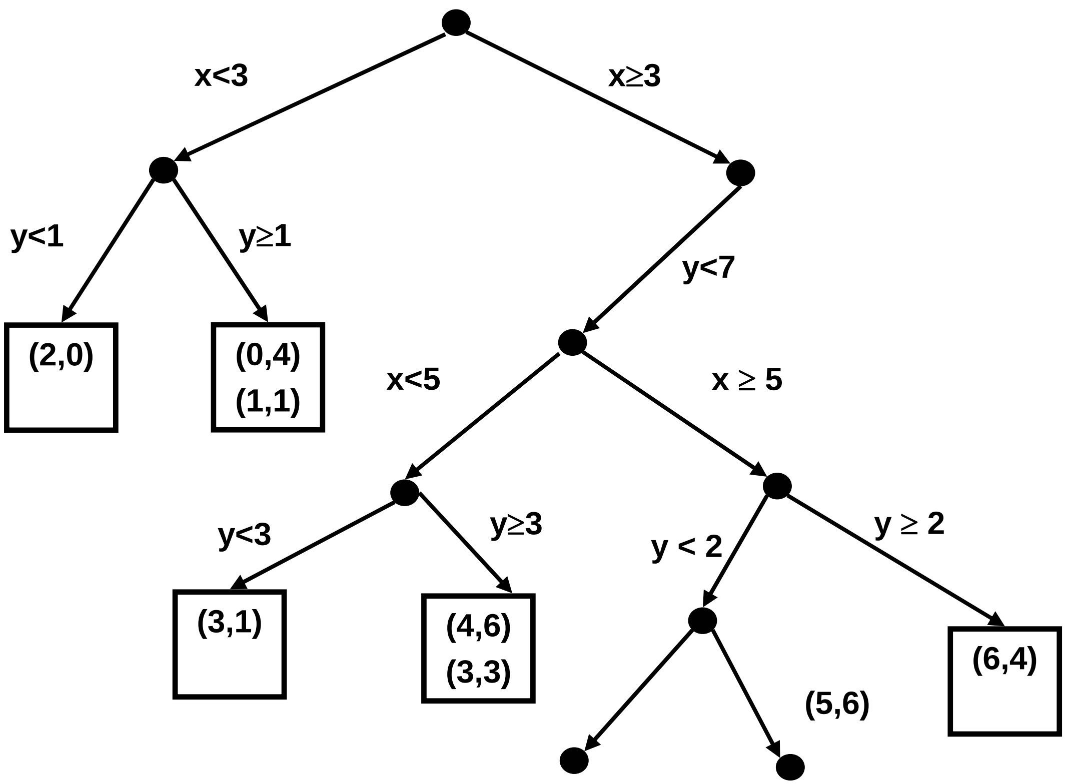
Prof. Kai-Uwe Sattler

Prof. Alfons Kemper, Dr. Eickler

Prof. Hector Garcia-Molina

# kd-Tree General Idea

- Binary, rooted tree
- Each inner node has two children
- Path is selected based on a pair (dimension / value)
- Dimensions need not be statically assigned to levels of the tree
  - Can be rotating, random, decided at time of block split, ...
  - Usually: rotating
- Data points are only stored in leaves
- Each leave stores points in a n-dimensional hypercube with m border planes (m≤n)

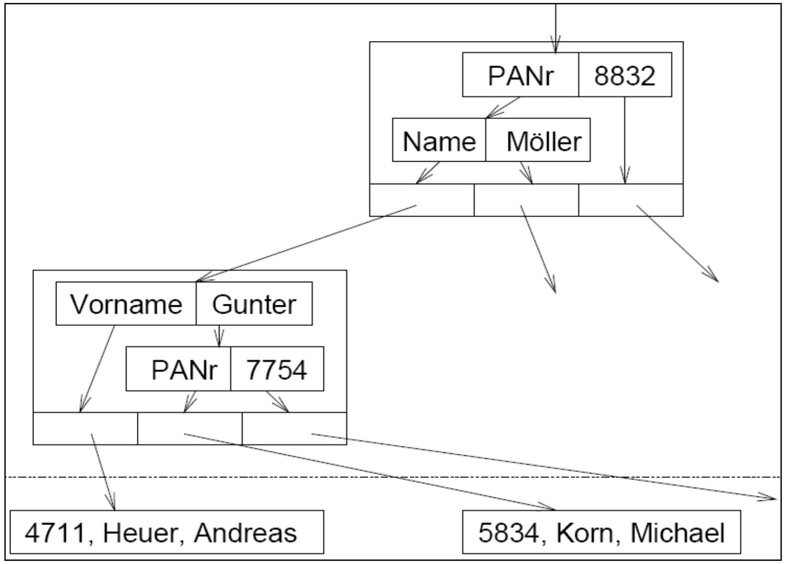


# kd-Tree Search Operations

- Exact point search
  - In each inner node, decide direction based on split condition
  - Search leaf for searched point
- Partial match query
  - If dimension of condition in inner node is part of the query – proceed as for exact match
  - Otherwise, follow all children in parallel
    - Leads to multiple search paths
- Range query
  - Follow all children matching the range conditions
    - Again: multiple search paths
- Nearest Neighborhood
  - Chose likely “close-enough” range and perform range query
  - If no success, iteratively broaden range

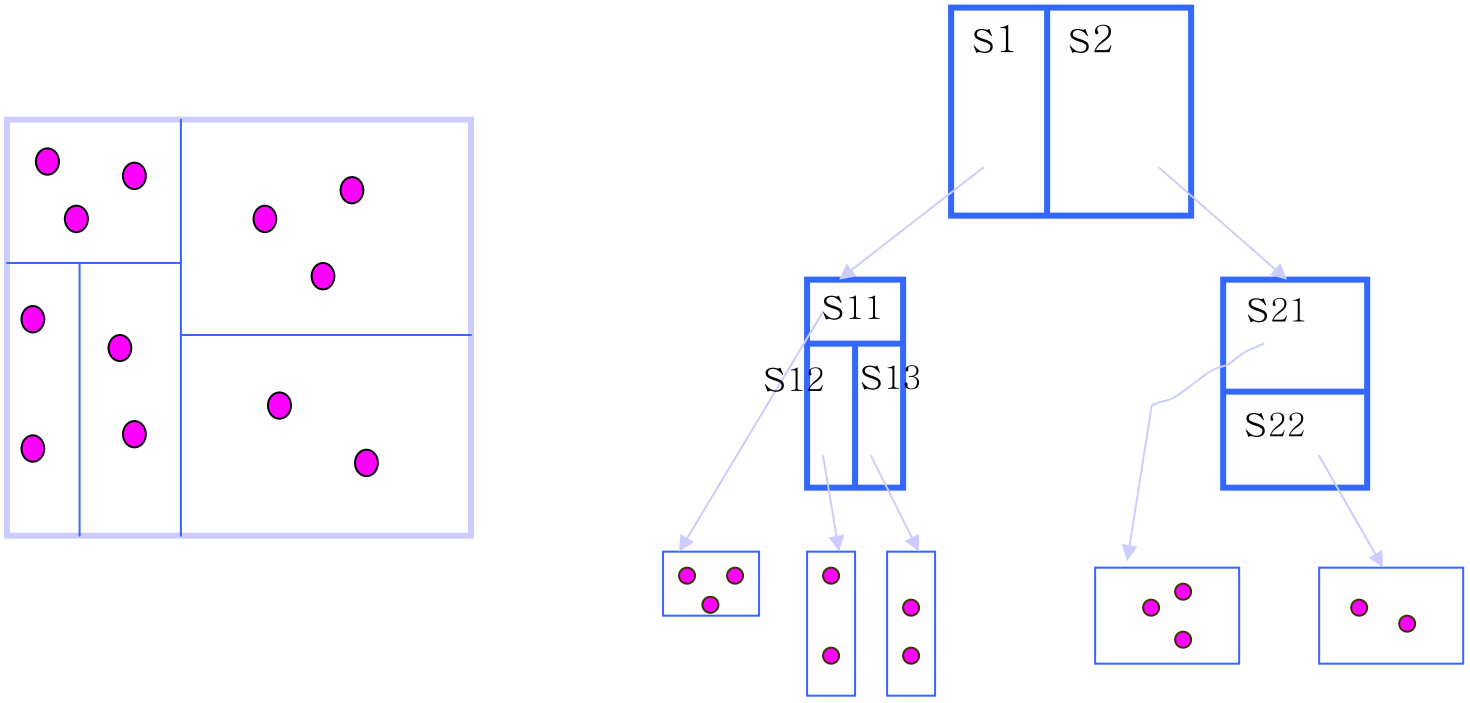
# kdb trees

- Option 2: Store entire subtrees in one block
  - Inner nodes still have only two children
    - But those are (usually) stored in the same block
    - We need to “map” nodes to trees
  - kdb-tree: inner nodes store kd-trees
- Operations
  - Searching: As with kd trees
    - But on average better IO complexity
  - Insertion/Deletion
    - Complex schemes for keeping balance in tree (later)



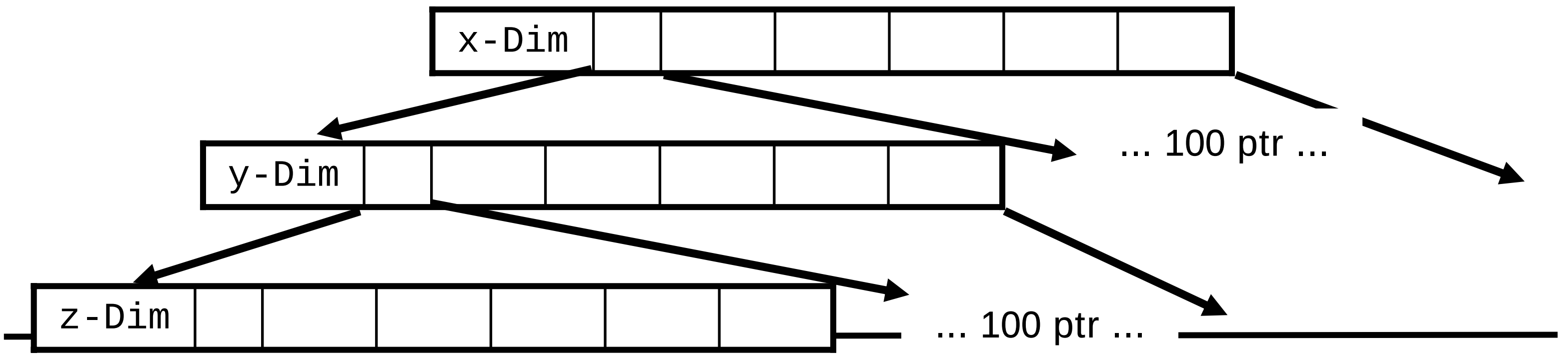
# Another View

- Inner nodes define (possibly open) bounding boxes on subtrees
- kdb tree is a hierarchical index structure



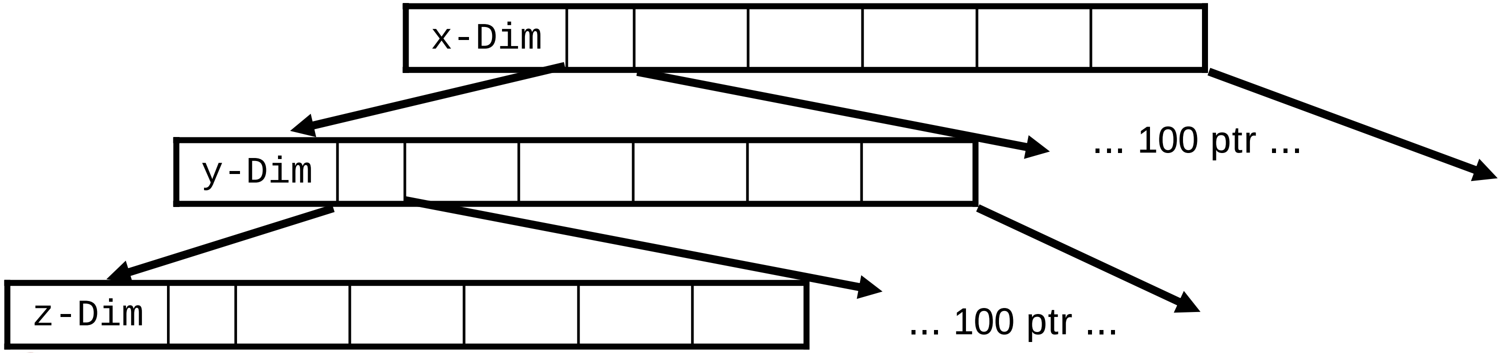
# Example – Composite Index

- Consider 3 dimensions, n=1E7 points, block size 4096, |point|=9, |b-ptr|=10
  - We need ~22000 leaf blocks
- Composite B* index
  - Inner blocks store at least 100 pointers (max ~220)
  - We need 3 levels (2nd level has 10.000 pointers)
  - With uniform distribution, 1st level will mostly split on 1st dimension, 2nd level on 2nd dimension, 3rd level on 3rd dimension
- Box query in all three coordinates, 5% selectivity in each dimension
  - We read 5% of 2nd level blocks = 5 IO
  - For each, we read 5% of 3rd level blocks = 5*5=25 IO
  - For each, we read 5% of data blocks = 125 IO
  - Altogether: 155 IO



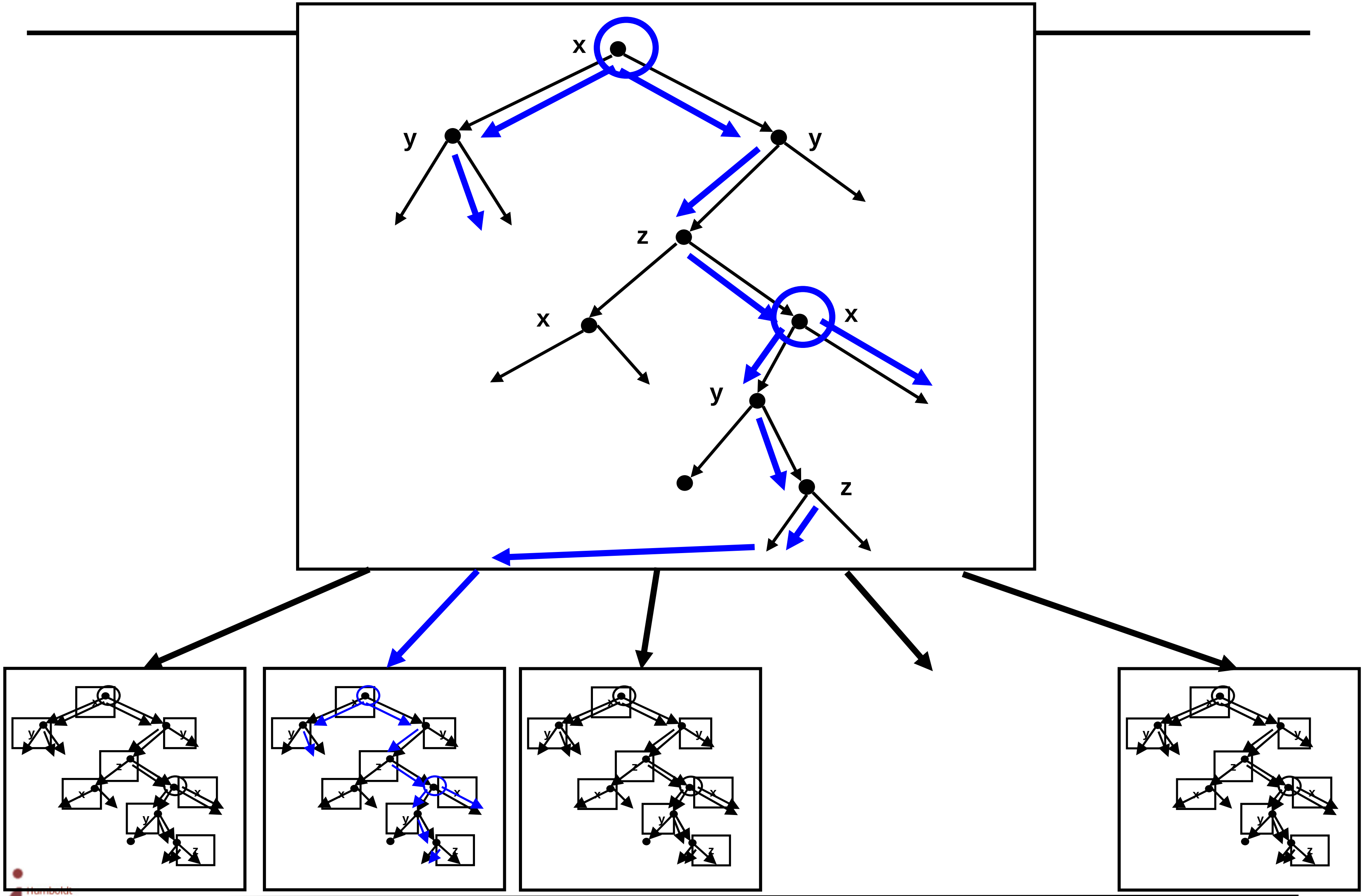
# Example Partial Box Query

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  - We need to scan all 100 2nd level blocks
    - Each 2nd level block contains the 5% range
  - Next, we scan 5% of 3rd level blocks = 500 blocks
    - We follow 5% of pointers from 2nd level blocks
  - For each, we read 5% of data blocks = 2500 data blocks
  - Altogether: 3100 IO
- Note: 5% selectivity in 2 dims means 0.0025 selectivity altogether = 25000 points
  - Only 60 blocks if optimally packed



# Example – kdb-tree

- Balanced tree will have ~14 levels
  - ~400 points in one block (assume optimal packaging)
  - We need to address 1E7/400 = 25.000~$2^{14}$ blocks
- Consider 128=$2^7$ inner nodes in one block
  - Rough estimate; we need to store 1 dim indicator, 1 split value, and 2 b-ptr for each inner node, but most b-ptr are just offsets into the same block
- kdb tree structure
  - 1st level block holds 128 inner nodes = levels 1-7 of kd tree
  - There are 128 2nd level blocks holding levels 8-14 of kd tree
- 1st block evenly splits space in 128 regions
- Box query in all three coordinates, 5% selectivity in each dimension
  - Overall selectivity is $(0.05)^3$ = 0,000125% of all points (1250 points)
  - Very likely, we need to look at only one 2nd level block
    - On 7 levels, 2 dim. will have been split into 4 sections, one dim. into 8
    - If query intersects with split points in each dimension: worst case 8 IO
    - Example: 100 values in one dimension, split will be at 25,50,75, query region covers 5 consecutive values – only 30 of 95 such regions cross a split point
  - Very likely, we need to look at only 4 data blocks (holding together the 1250 points)
  - Altogether: 1+1+4 = 6 IO (compared to 155 for composite index)



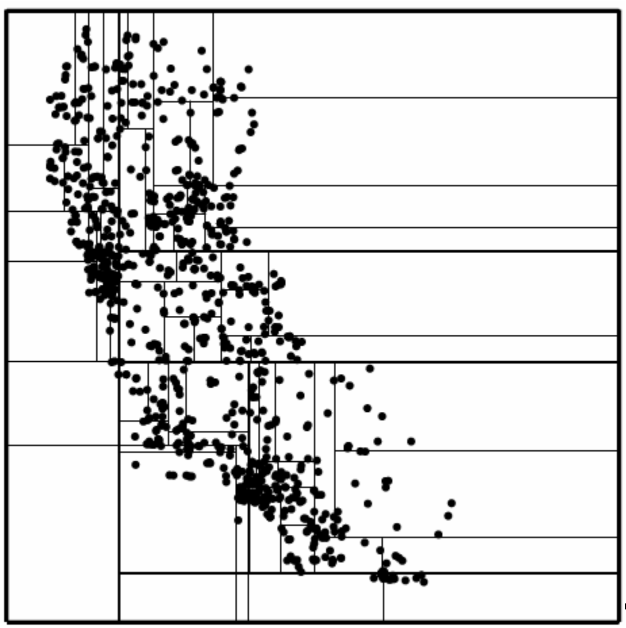
x
y
y
z
x
x
y
z

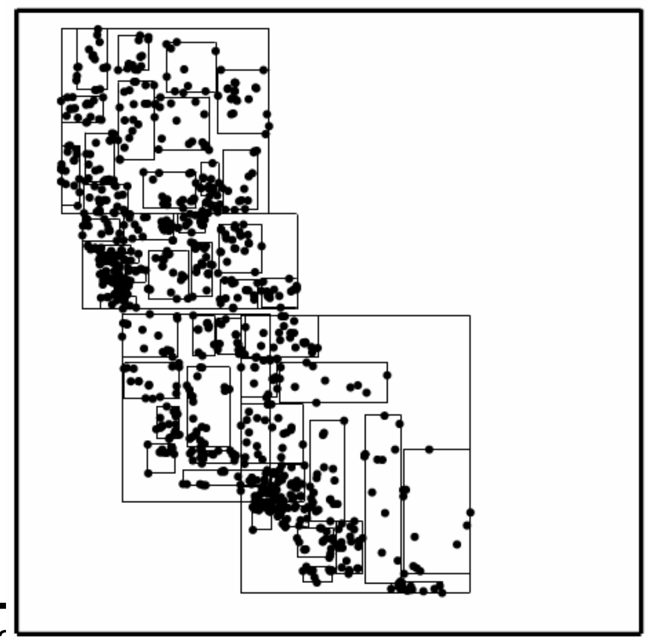
# Example - Partial Box Query 2

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  - In first block (7 levels), we have ~2 splits in each dimension
    - Two times 2 splits, one time three splits
    - Assume we miss the dimension with 3 splits
  - Hence, in ~4 of 7 splits we know where we need to go, in ~3 splits we need to follow both children
  - We need to check only $2^3=8$ second-level blocks
    - Again – number gets higher when query range crosses split points
  - Same argument holds in 2nd level blocks = 8*8 data blocks
  - Altogether: 1+8+64 = 73 IO
    - We almost reach optimum with 60 blocks
    - Compare to 3100 for composite index
- Beware
  - We made many, many assumptions
  - Layout of subtrees to nodes rarely optimal
  - Optimal packaging of points in blocks not realistic for real data
  - Performance can greatly vary due to n# of dimensions, distributions, order of insertions and deletions, selectivity, split and merge policies, ...

# R-Trees

- Can store geometric objects (with size) as well as points
- Each object is stored in exactly one region on each level
- Since sized objects may overlap, R tree regions may overlap
- Better adaptation to distribution of data objects
- Only those hyperregions containing data objects are represented
- Many variations (see literature)





# General Idea

- R-trees store n-dimensional rectangles
  - For geometric objects, use minimal bounding box (MBB)
- Objects in a region of the n-dimensional space are stored in a block
  - The region borders is the MBB of all objects in contains
  - Regions may overlap – see below
- Regions are recursively contained in larger regions
  - Tree-like shape
  - Region borders in each level are MBB of all child regions
  - Regions are only as large as necessary
  - Regions of a level need not cover the space completely
- Regions in one level may overlap
  - Or not – variation of classical R tree
  - Without overlaps: much more complicated insertion/deletion, but better search complexity
- Finding all rectangles overlapping with a query rectangle
  - In each level, intersect with all regions
  - More than one region might have non-empty overlap
    - All must be considered
    - In general, no O(log(n)) complexity

# Inserting an Object

- In each level, find regions that contains object
  - Completely or partly
  - More than one region with complete overlap
    - Chose one (smallest?) and descend
  - None with complete, but several with partial overlap
    - Chose one (largest overlap?) and descend
  - No overlapping region at all
    - Chose one (closest?) and descend
  - We insert object in only one region
- In leaf node with space available
  - Insert object and adapt MBB
  - Recursively adapt MBBs up the tree
  - This generates larger and larger overlaps – search degrades
- In leaf node with no space available
  - Split block in two regions
  - Compute MBBs
  - Can affect MBB of higher regions – ascend recursively

# Block Splits

- Problem: How should we optimally split a overflow-node into two regions?
- Option 1: Avoid overlaps, cover large space
  - Compute partitioning such that there exists a separating hyperplane
  - Minimizes necessity to descend to different children during search
  - Generally requires larger regions – search in empty regions is detected later
- Option 2: Allow overlaps, minimize space coverage
  - Compute partitioning such that sum of volumes of MBBs is minimal
  - Overlaps increases changes to descend on multiple paths during search
  - But: Unsuccessful searches can stop early

# Block Splits

- Whatever strategy we chose
  - Consider a block with n objects
  - There are $2^n$ possibilities to partition this block into two
  - Most strategies require to check them all
  - Use heuristics instead of optimal solution
- R* tree
  - Chose as criterion combination of sum of covered spaces, space of intersection, and sum of girt
  - Use heuristic for concrete decision
  - Currently best strategy (still?)

# Multidimensional Data Structures Wrap-Up

- We only scratched the surface
- Partitioned Hashing, Gridfile, kdb-Tree, R-Tree
- Other: X tree, hb tree, R+ tree, UB tree, ...
  - Store objects more than once; other than rectangular shapes; map coordinates into integers; ...
- Curse of dimensionality
  - Your intuition plays tricks on you
  - The more dimensions, the more difficult
    - Balancing the tree, finding MBBs, split decisions, etc.
  - All structures begin to degenerate somehow
    - Exploding size of directories, linear kdb-trees, all regions overlap, ...
  - Often, linear scanning of objects is quicker
    - Or: Compute lower-dimensional, relationship-preserving approximations of objects and filter on those

# Content of this Lecture

- Relational operations
- Physical query plan operators
- Implementing (some) relational operators

# 5 Schichten Architektur

Wir sind hier:

Mengenorientierter Zugriff

**Datenmodellebene** – Anfrageübersetzung, Zugriffspfadwahl, Zugriffskontrolle, Integritätskontrolle

…entierter Zugriff

**Logischer Zugriff** – Sortierung, Transaktionsverwaltung, Cursorverwaltung

Interne Satzschnittstelle

**Speicherstrukturen** – Record Manager, Index Manager, Sperrverwaltung, Log / Recovery

Systempufferschnittstelle

**Pufferverwaltung** – Speichermanagement, Puffermanagement, Caching-Strategien

Dateischnittstelle

**Betriebssystem** – Externspeicherverwaltung

Geräteschnittstelle

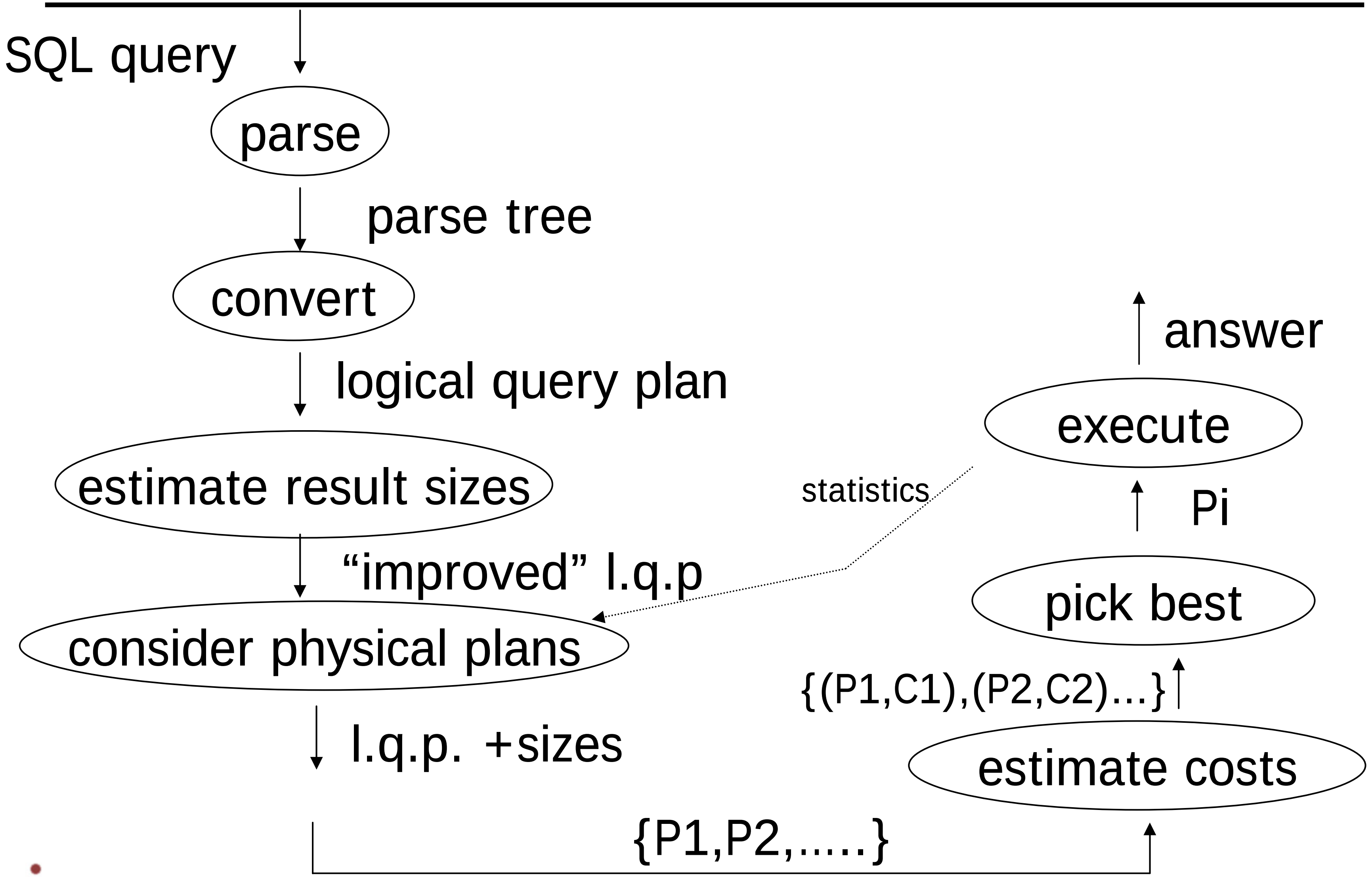
# Query Execution

- We have
  - Structured Query Language SQL
  - Relational algebra
  - How to access tuples in many ways (scan, index, ...)
- Now
  - Given a SQL query
  - Find a clever way and order of accessing tuples such that the answer to the query is computed
    - Usually, we won't find the best way, but avoid the bad
  - Use knowledge about value distributions, access paths, query operations, IO cost, ...
  - Compile a declarative query in a good executable (procedural) program

# Query Execution

- Steps (rough sketch)
  - Translate SQL query in relational algebra term
  - Logical optimization
    - Each term can be rewritten in many other, semantically equivalent terms
    - For each operator we have multiple implementations
    - Choose the hopefully best query plan (= term)
  - Physical optimization
    - For each relational operation, we have multiple possible implementations
    - Table access: scan, different indexes, sorted access through index, ...
    - Joins: Nested loop, sort-merge, hash, index, ...
  - Query execution
    - Execute the best query plan found

# Complete Workflow

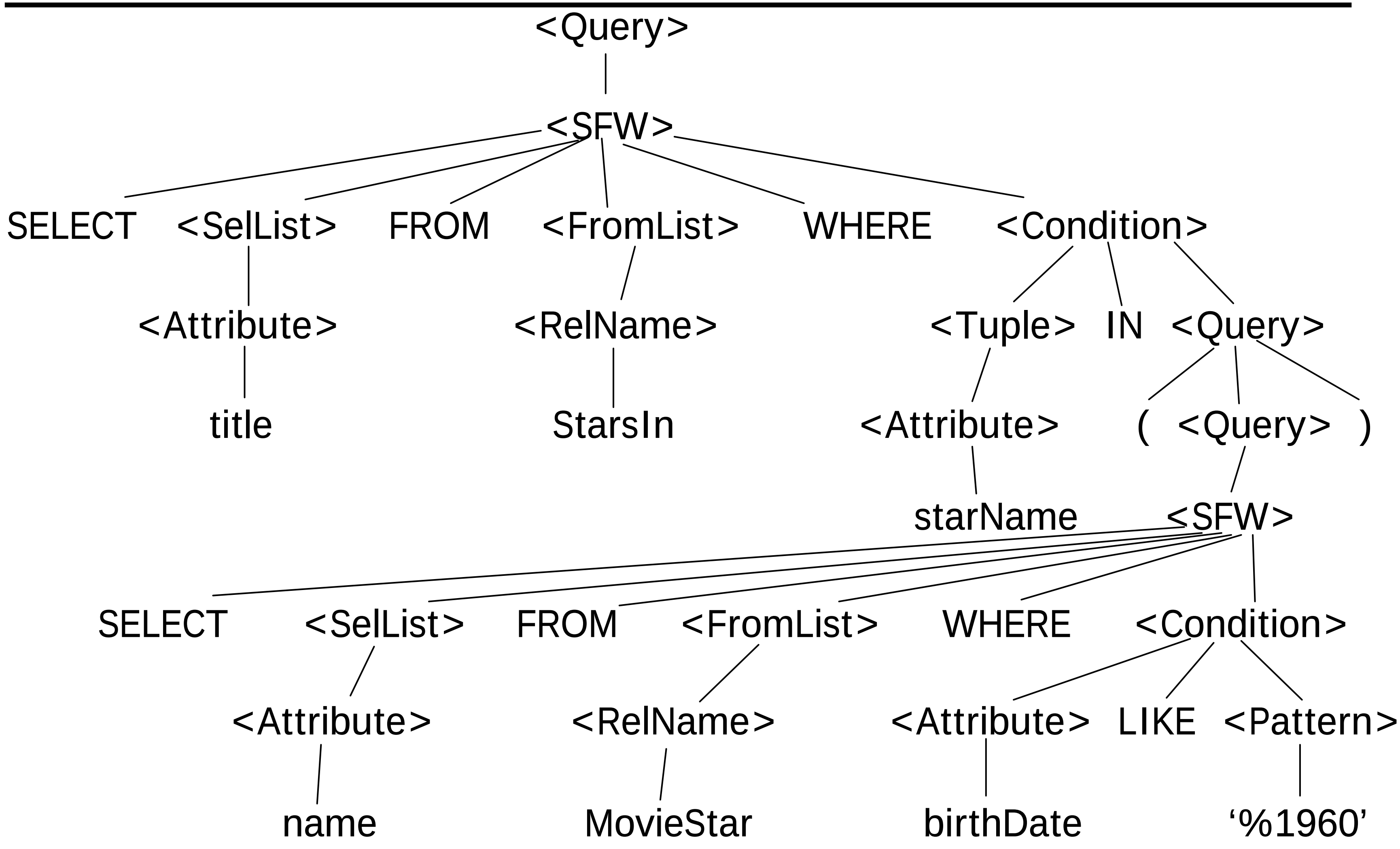


# Example SQL query

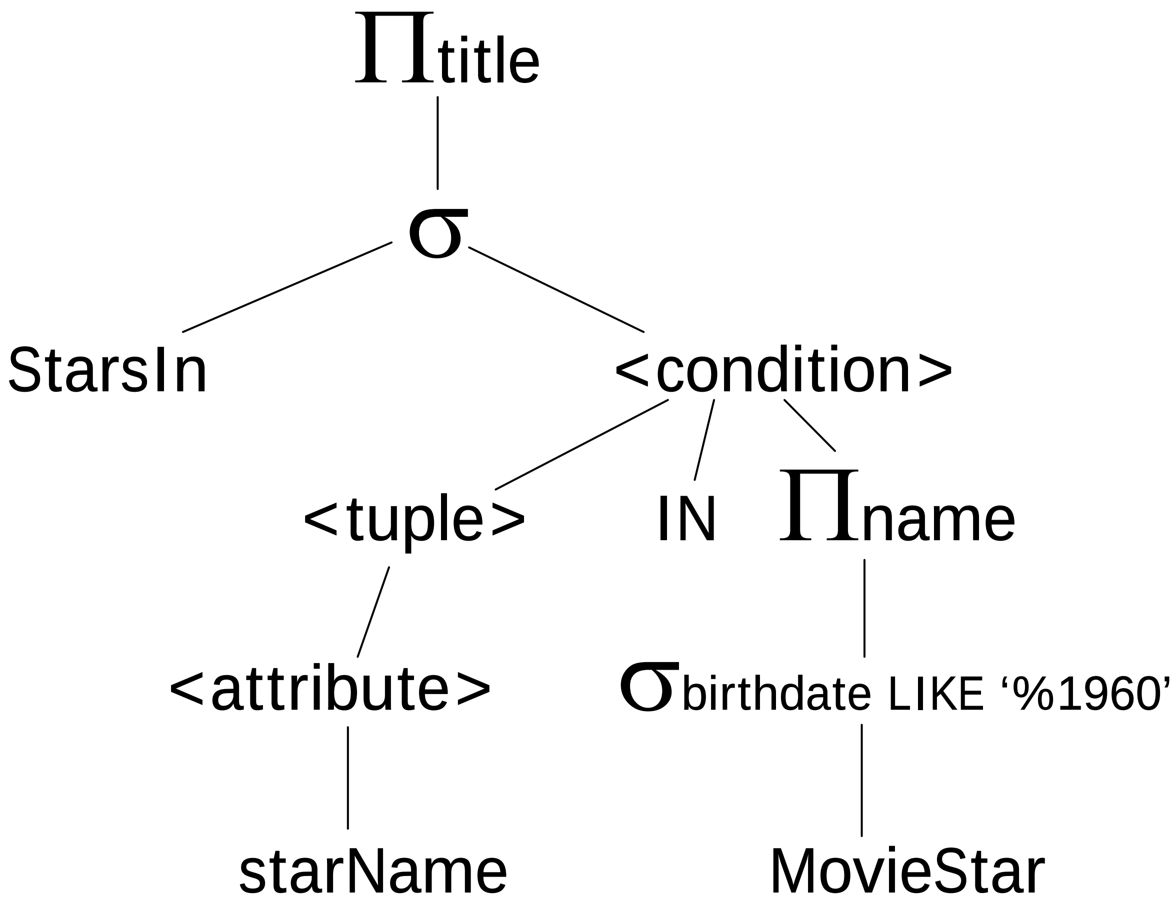
```
SELECT title
FROM StarsIn
WHERE starName IN (
        SELECT name
        FROM MovieStar
        WHERE birthdate LIKE ‘%1960’
);
```

(Find the movies with stars born in 1960)

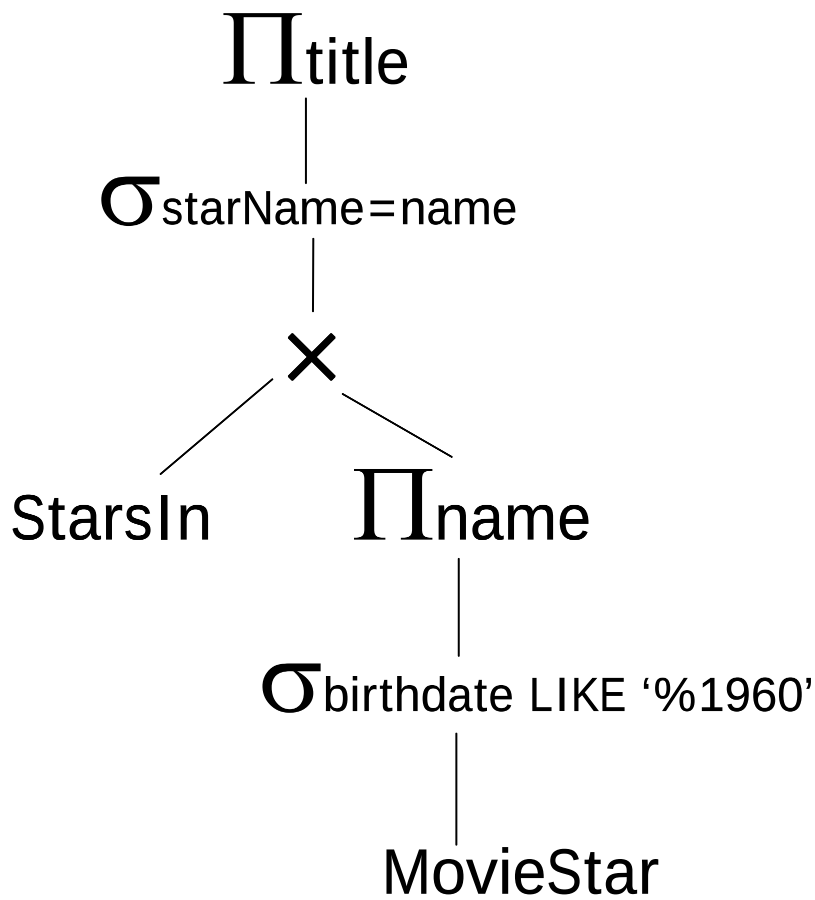
# Parse Tree


<Query>
<SFW>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
<RelName>
<Tuple>
IN
<Query>
title
StarsIn
<Attribute>
(
<Query>
)
starName
<SFW>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
<RelName>
<Attribute>
LIKE
<Pattern>
name
MovieStar
birthDate
‘%1960’

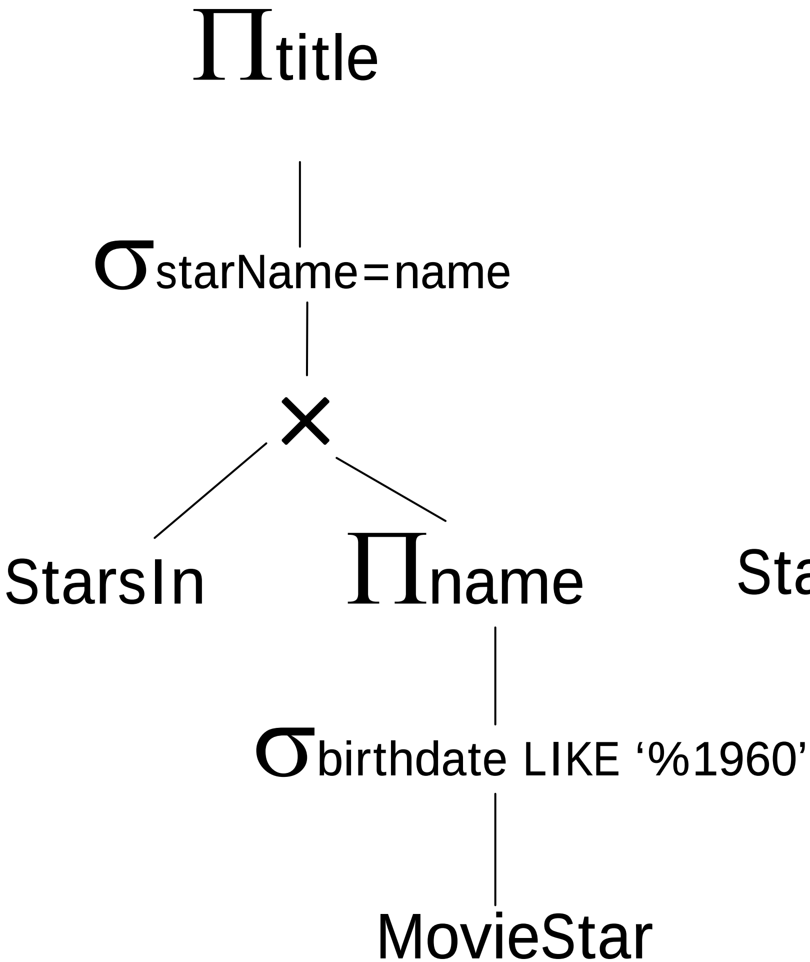
# From Parse Tree to Relational Algebra

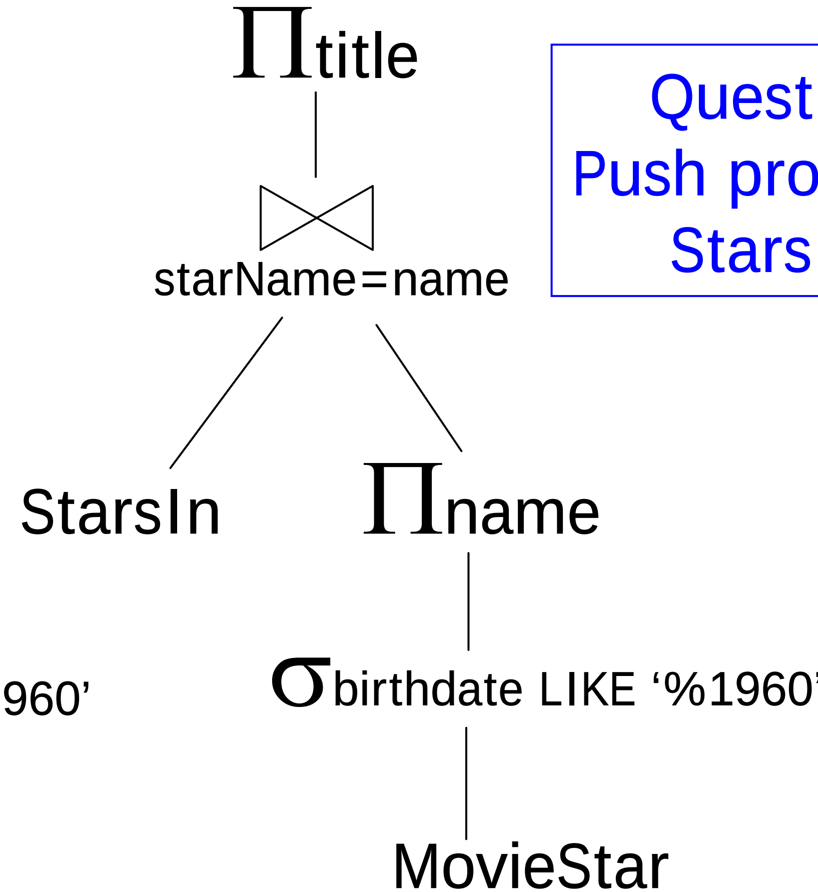


# Relational Algebra Term as Tree: Logical Query Plan



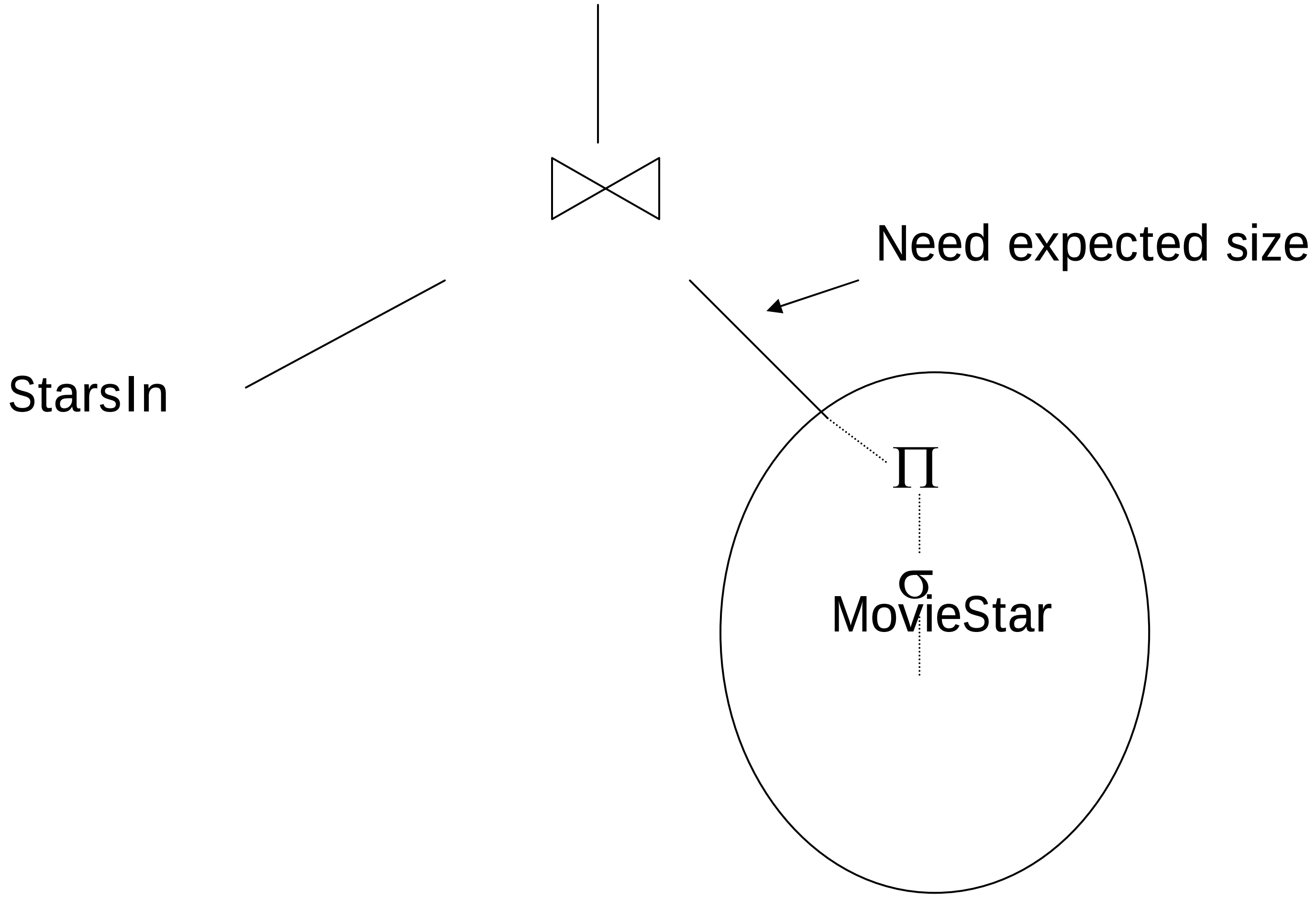
# Improved Logical Query Plan



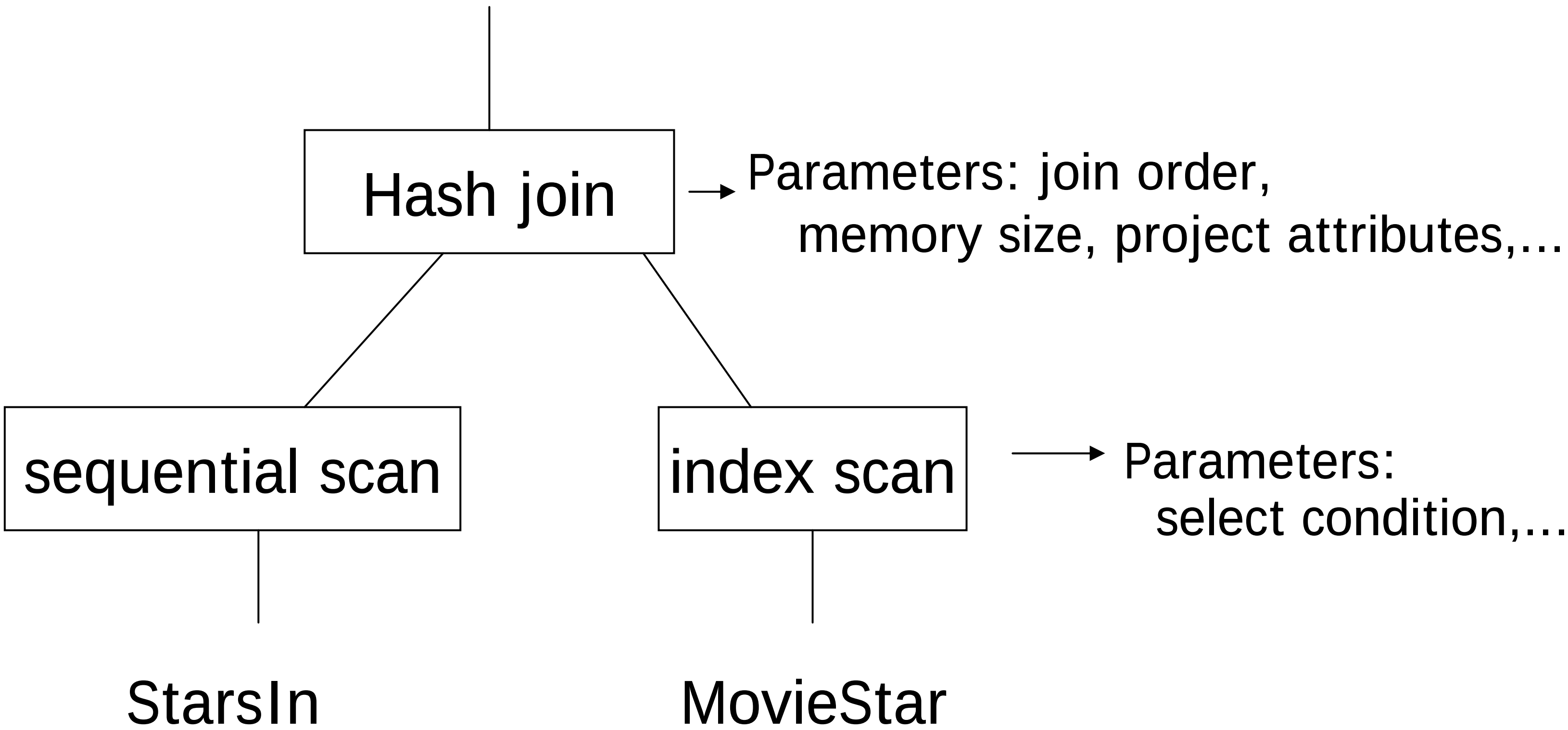




Question:
Push project to StarsIn?

# Estimate Result Sizes



# Physical Plan



# Estimate costs

L.Q.P

P1 P2 .... Pn

C1 C2 .... Cn

Pick best!

# Relational Operations: One Table

- In the following: Table means table or intermediate result
- One table operations
  - Selection $\sigma$
    - Read table and filter away tuples based on condition
    - Possibility: Use index to access only the qualifying tuples
  - Projection $\pi$
    - Read table and remove attribute values (columns)
    - In SET semantic, also duplicates must be filtered
    - Projection usually decreases size of table
      - When not??
  - Grouping
    - Read table and build structure on grouping attribute(s)
    - "Aggregate" (or remove) other columns
  - Duplicate elimination (DISTINCT)
  - Sorting
    - Not an operation in relational algebra
    - But very helpful for physically implementing relational operations

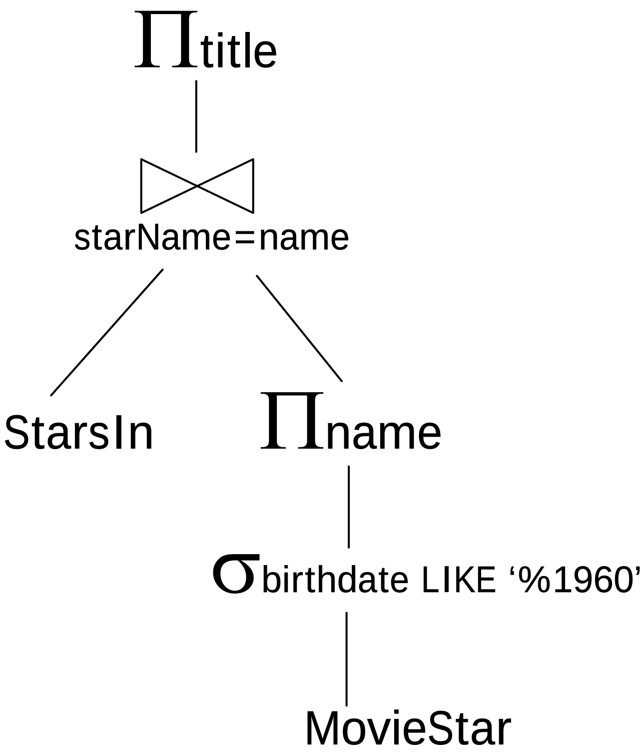
# Relational Operations: Two Tables

- Two table operations
  - Cartesian product x
    - Usually avoided – combine product and selection to join
      - Products in a plan are hints to wrong queries
  - Derived operation: Join $\bowtie$
    - Read two tables (in whatever order), find matching tuples
    - Natural join, theta join, equi join, semi join, outer join
    - Nested-loop join, sort-merge join, hash join, index join, ...
  - Union $\cup$
    - Read two tables and build union
    - Might include duplicate elimination
  - Intersection $\cap$
    - Same as join over all attributes
  - Minus $/$
    - Subtract tuples of one table from tuples from the other

# Query Execution

- Assume that a query plan has been chosen
- Each relational operation needs a physical implementation
  - Chose best if there are many
  - Choice often has “side-effects” – sorted results, pipelining, ...
  - Hence, choices should not be made independently of plan generation and choices for other operators
- Iterator concept
  - Each operator implementation offers three methods
  - Open, next, close
- Two modes of iterators calling each other
  - Blocked
  - Pipelined

# Example - Blocked

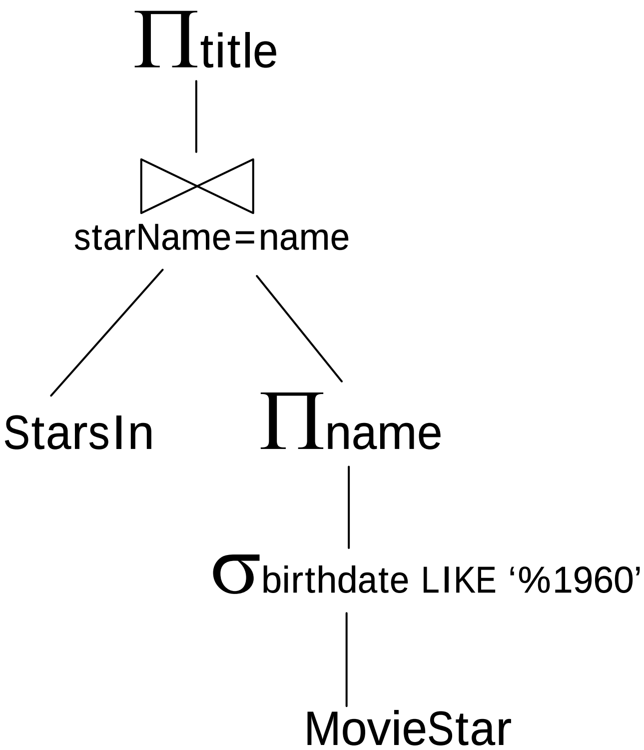



```
P= projection.open();
while p.next(t)
    output t.title;
p.close();
```

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

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

# Example - Pipelined




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

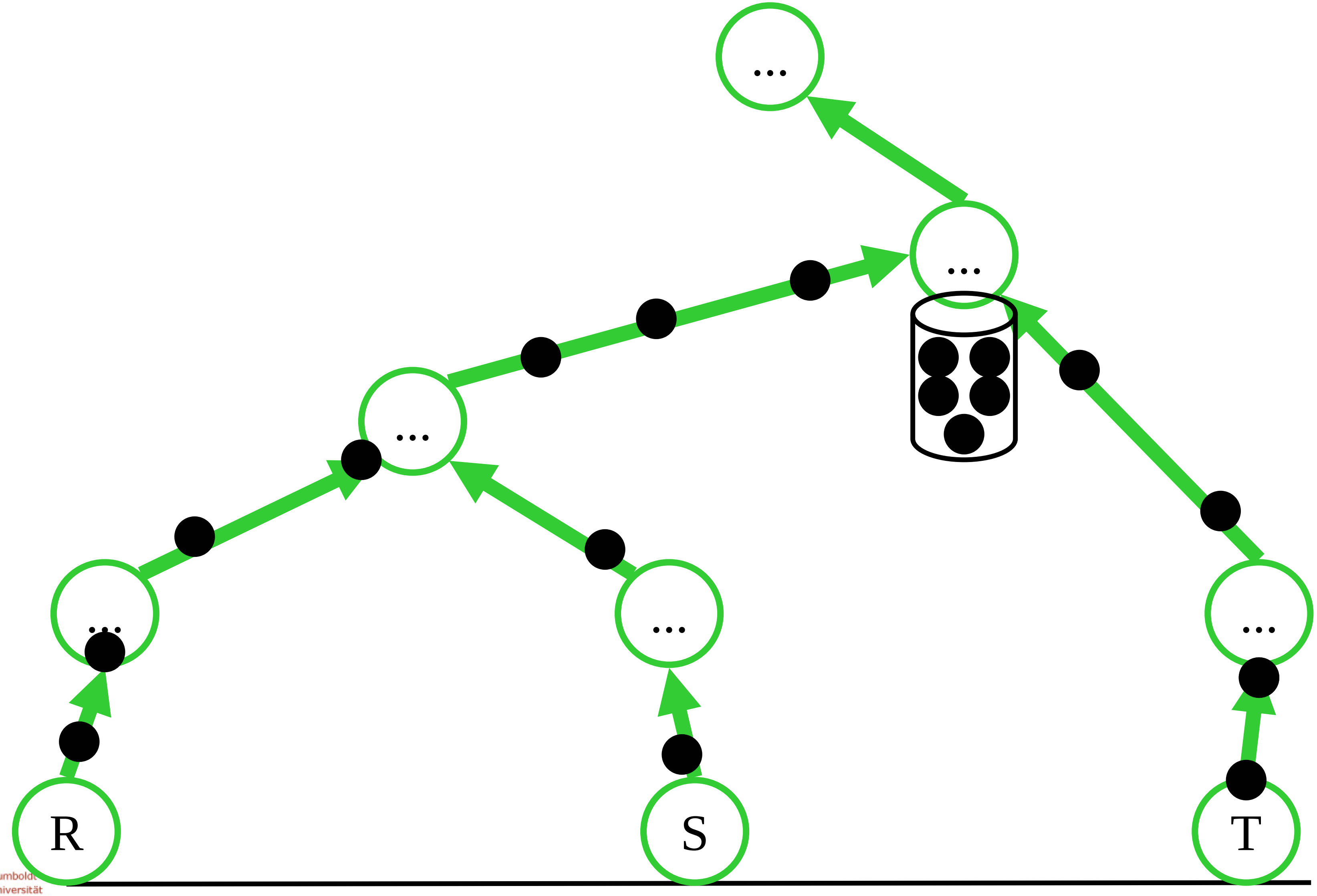
```
Class projection {
open() {
  j = join.open();
}
next(t) {
  return j.next( t);
}
close() {
  j.close();
}
}
```

```
Class join {
Open() {
  l = table.open();
  r = projection.open()
  l.next( tl);
}
next( t) {
  if r.next(tr)
    if l.starname=r.name
      t=tl⋈tr;
      return true;
  else
    if l.next( tl)
      r.reset();
      return next( t);
    else
      return false;
}
close() {
  l.close;
  r.close();
}
}
```

# Pipelined versus Blocked

- Pipelining is in general highly advantageous
  - No need for real buffering
    - When intermediate results are large, buffers need to be stored on disk
  - Operations can be distributed to different threads or CPUs
- Pipeline breaker
  - Some operations cannot be pipelined
  - Sorting: next() can be executed only after entire table was read
    - Exception: When input is sorted, e.g., from previous operation
  - Grouping and aggregation
    - Usually realized by first sorting or hashing (later)
      - To avoid larger buffers for intermediate results
    - Then, next() performs aggregation for one group and returns
  - Minus, intersection
- Projection with duplicate elimination
  - Need not be pipeline breaker
  - next() can return early (no sorting required)
  - But we need to keep track of all values already returned – requires large buffer

# Pipeline Breaker



# Bag and Set Semantic

- Relational algebra has SET semantic
  - All relations are duplicate-free
  - Result of each query is duplicate-free
  - Result of each intermediate result is duplicate-free
- SQL databases use BAG semantic
  - More practical in applications
  - Usually, PKs prevent existence of real duplicates
  - Note: Removing duplicates in SQL is not trivial (how??)
- This makes many things easier
  - Duplicate elimination can be avoided
- But: Duplicate elimination is still a topic
  - DISTINCT clause
  - What else??

# Select and Update

- We do not discuss update, delete, insert
  - Update and delete have queries – “normal” optimization
    - But: data tuples must be loaded (and locked and changed)
    - Some tricks don’t work any more (e.g. “oversized” index)
  - Insert may have query
- Interference
  - “Halloween” problem
  - Execute the following naively using an iterator on an index on salary
    - Give employees a raise
    - `UPDATE salary SET salary=1.1*salary`
  - What happens??

# Implementing Operations

- Most single table operations are rather straight-forward
  - See book by Garcia-Molina, Ullmann, Widom for detailed discussion
- Joins are more complicated – later
  - In general, binary operations are more complex
  - We will see some
- Sorting, especially for large tables, is complicated
  - External sorting – we have seen Merge-Sort
  - See textbooks on Algorithms and Data Structures
- We sketch three single table operations
  - Scanning a table
  - Duplicate elimination
  - Group By

# Scanning a Table

- At the bottom of each operator tree are relations
- Performing open-next-close means scanning the table
  - If table T has b blocks, this costs b IO
- Often better: combine with next operation
  - `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 with T.A=a
      - Index has height $\log_k(n)$
      - Accessing z tuples from T costs (worst-case) z IO
    - Complexity is identical O(n), but difference can be tremendous
      - Especially if A is a key, i.e., z=1
  - Projection: Integrate into table scan
    - Only easily possible for BAG semantic
    - Otherwise, a duplicate elimination step must be inserted

# Scanning a Table 2

- Selection 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
  - Using indexes
    - Compute TID lists for each conjunct
    - Intersect
    - Alternatives??
  - Without indexes
    - Scan table and evaluate condition for each tuple
- For complex conditions and small tables, linear scanning might be faster
  - Depends on expected result size
  - Cost-based optimization required (later)
    ```

# Duplicate Elimination

- Option 1: Use external sorting
  - Sort input table (or intermediate result) on DISTINCT columns
    - Can be skipped if table is already sorted
  - Scan sorted table and output only unique values
  - Generates output in sorted order
  - Pipeline breaker
- Option 2: Use internal sorting/hashing
  - Scan input table
  - Build internal data structure, holding each unique tuple once
    - Binary tree – some cost for balancing, robust
    - Hash table – might be faster, needs good hash function
  - When reading a tuple from the relation, check if it has already been seen
    - If no: insert tuple and copy it to the output; else: skip tuple
    - No pipeline breaker
    - Generates unsorted result
- How much IO will we need??

# Performance

- Assumptions
  - Main memory: m blocks
  - Table: b blocks
- Using external sorting
  - If table is sorted, we need b IO
  - If table not sorted, we need $2*b*\text{ceiling}(\log_m(b))+b$ IO
    - Improvable to $2*b*\text{ceiling}(\log_m(b))-b$ – how??
- Using internal sorting
  - If all distinct values fit into m, we need b IO
    - Estimate from statistics
  - Otherwise ... use two pass algorithms (e.g. hash-join like; later)
- What if DISTINCT column is key??

# Grouping and Aggregation

```
SELECT T.day_id, sum(amount*price)
FROM   Sales S
GROUP BY T.day_id
```

- Syntax
  - Select may contain only GROUP BY expressions and aggregate functions
- Semantics
  - Partition result of inner query according to the value of the GROUP BY attributes
  - For each partition, compute one result tuple: GROUP BY attributes and aggregate function applied on all 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??

Inner query → Partition → Aggregate → HAVING clause

# Implementing GROUP BY

- Proceed like duplicate elimination
- But we also need to compute the aggregated columns
  - No problem: SUM, COUNT, MIN, MAX, ANY
  - What to do for AVG??
  - What to do for Top5??
  - What to do for MEDIAN??

# Computing MEDIAN

- We need to consider all values for each group
  - Sort and chose middle one
- 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
- 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??