23CS404 – DATABASE SYSTEMS

UNIT V: QUERY PROCESSING AND QUERY OPTIMIZATION AND CASE STUDIES

Database System Concepts, 6th Ed.

LU40 - Indexing: B+ Tree

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

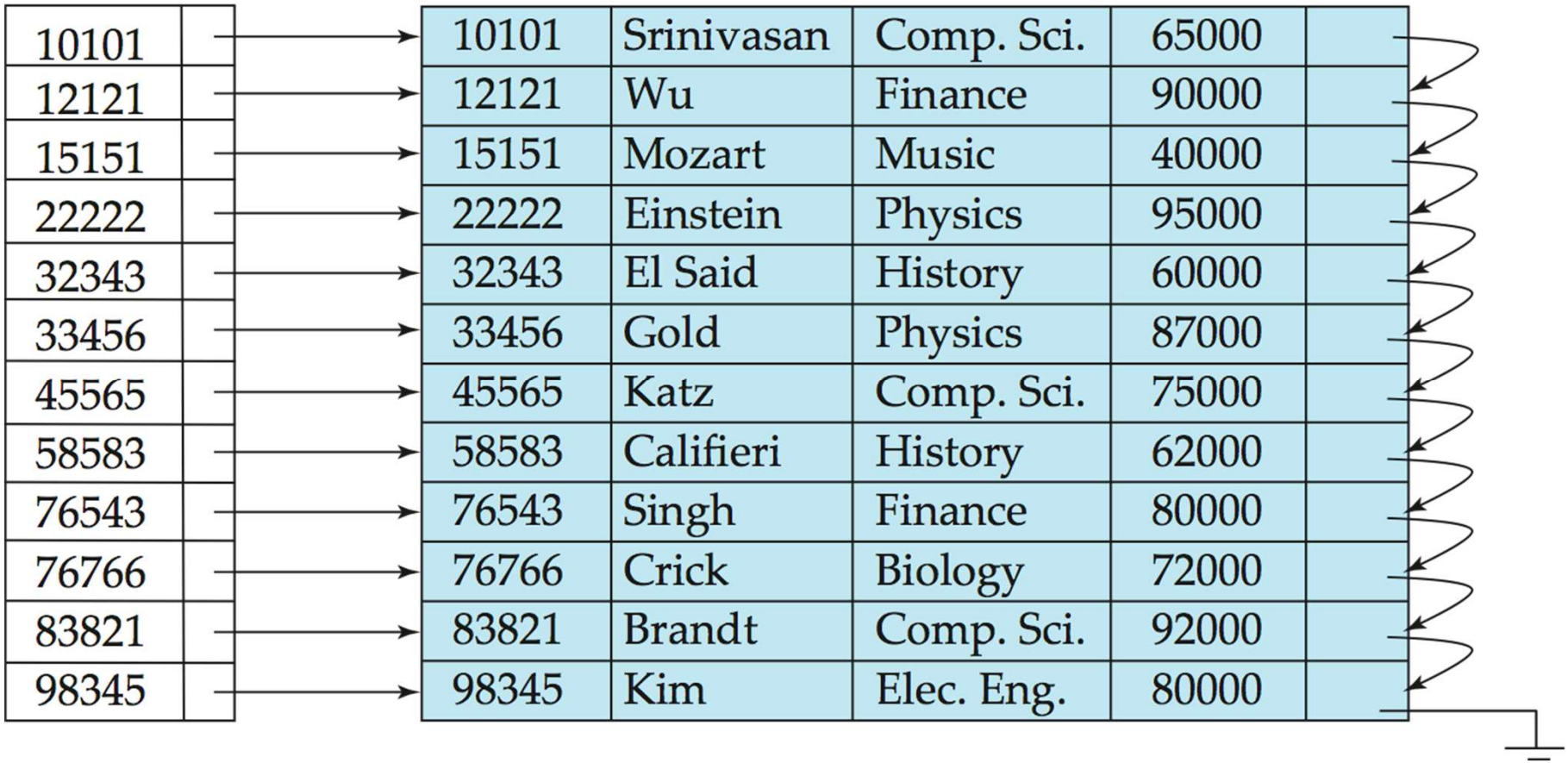
- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.

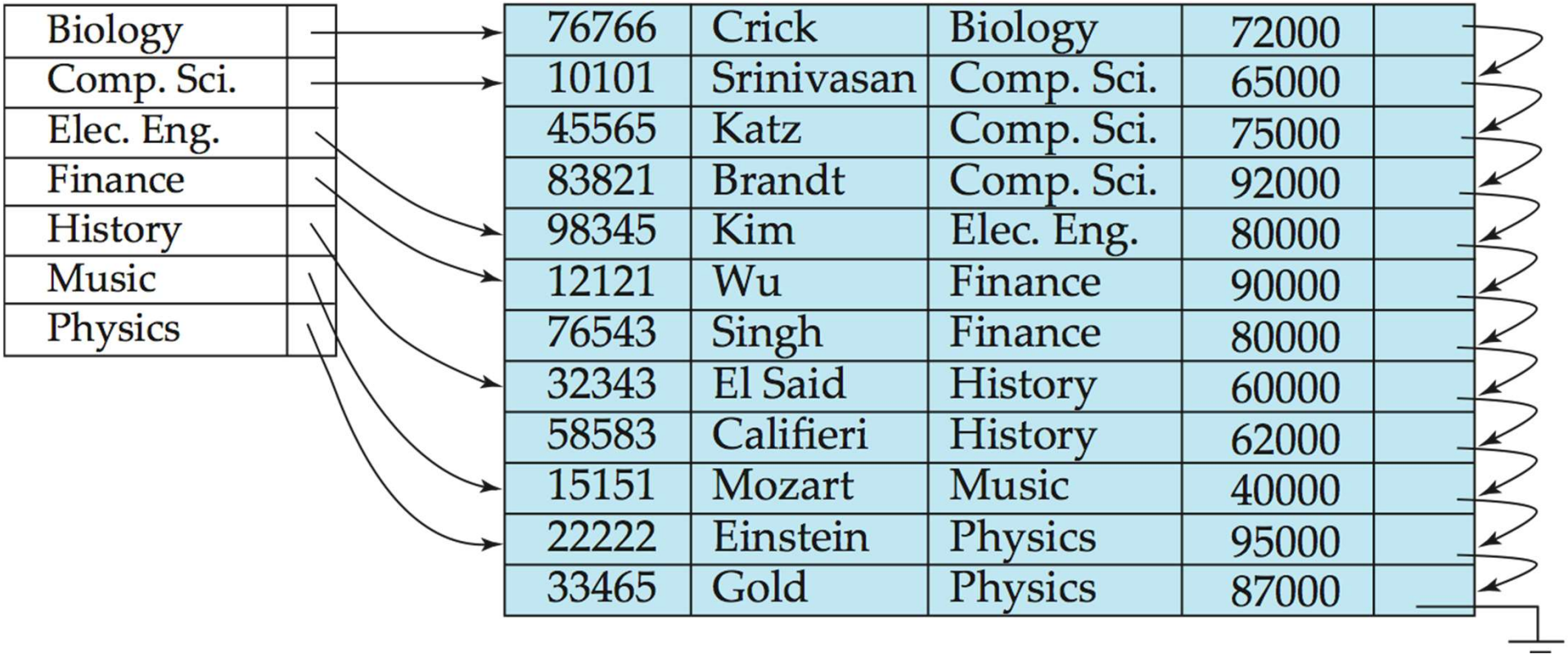
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



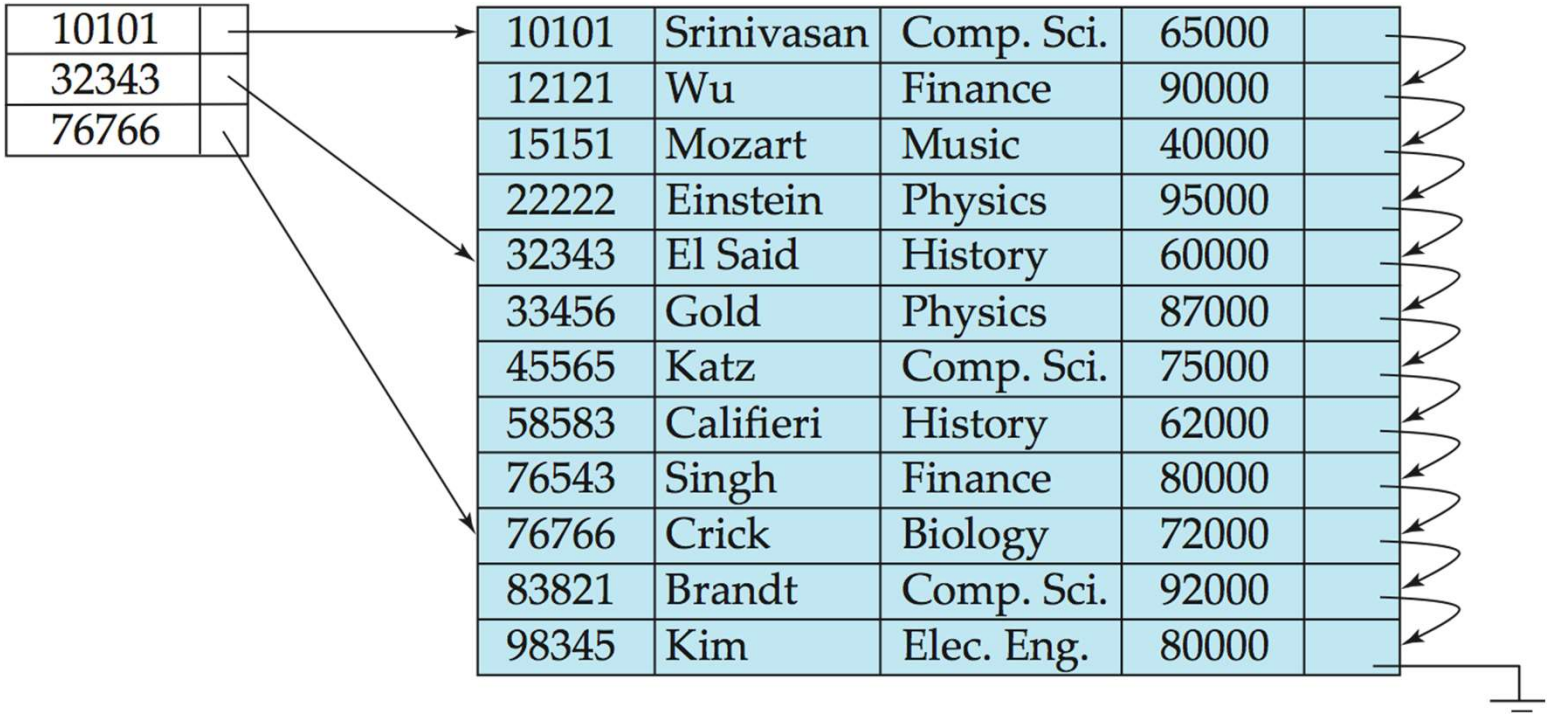
Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



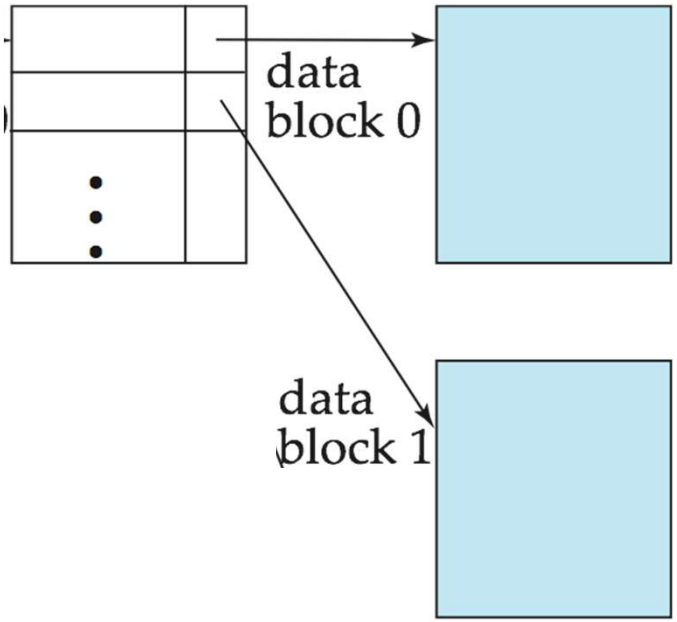
Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

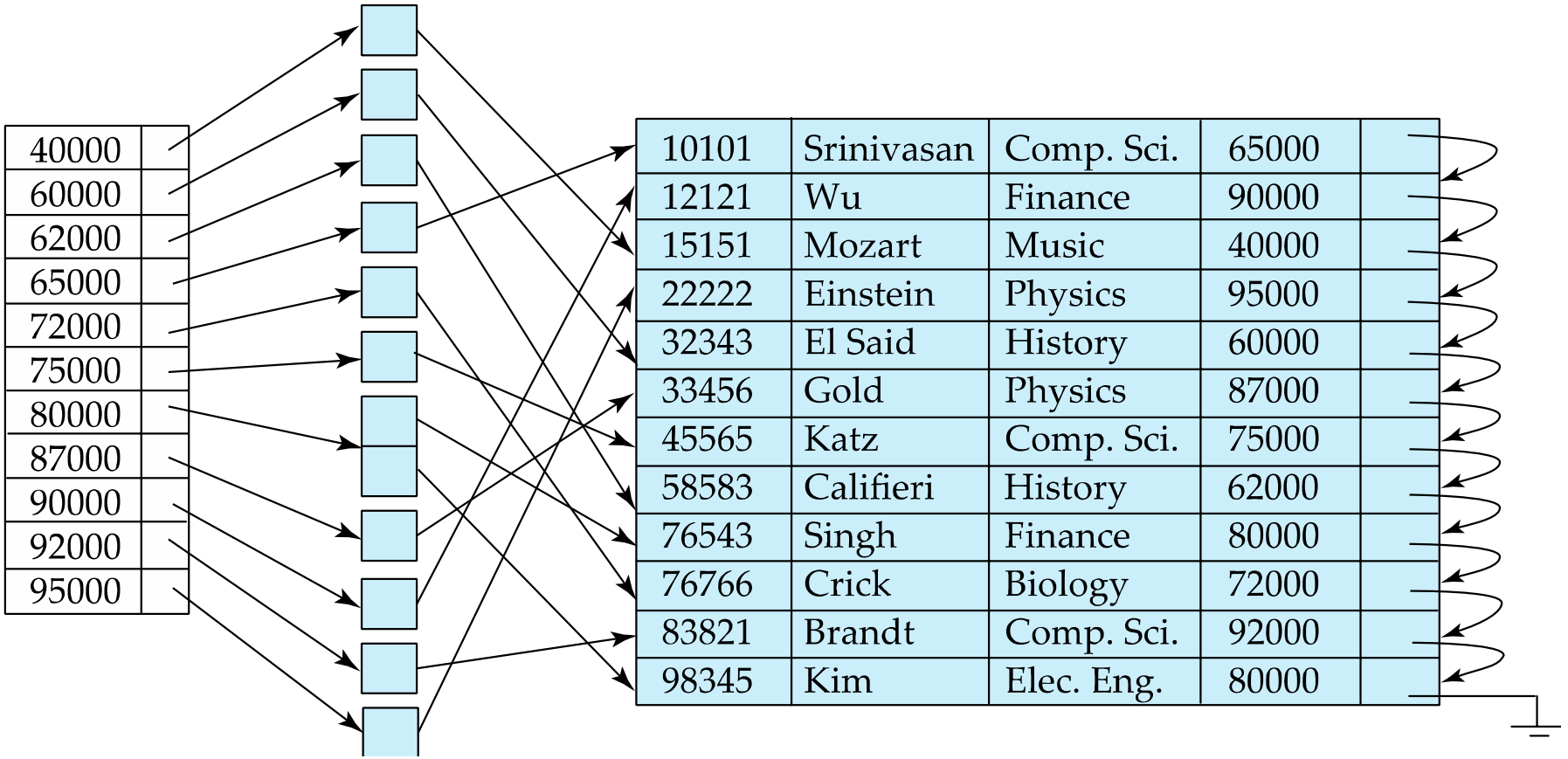


Sparse Index Files (Cont.)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices Example



Secondary index on *salary* field of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

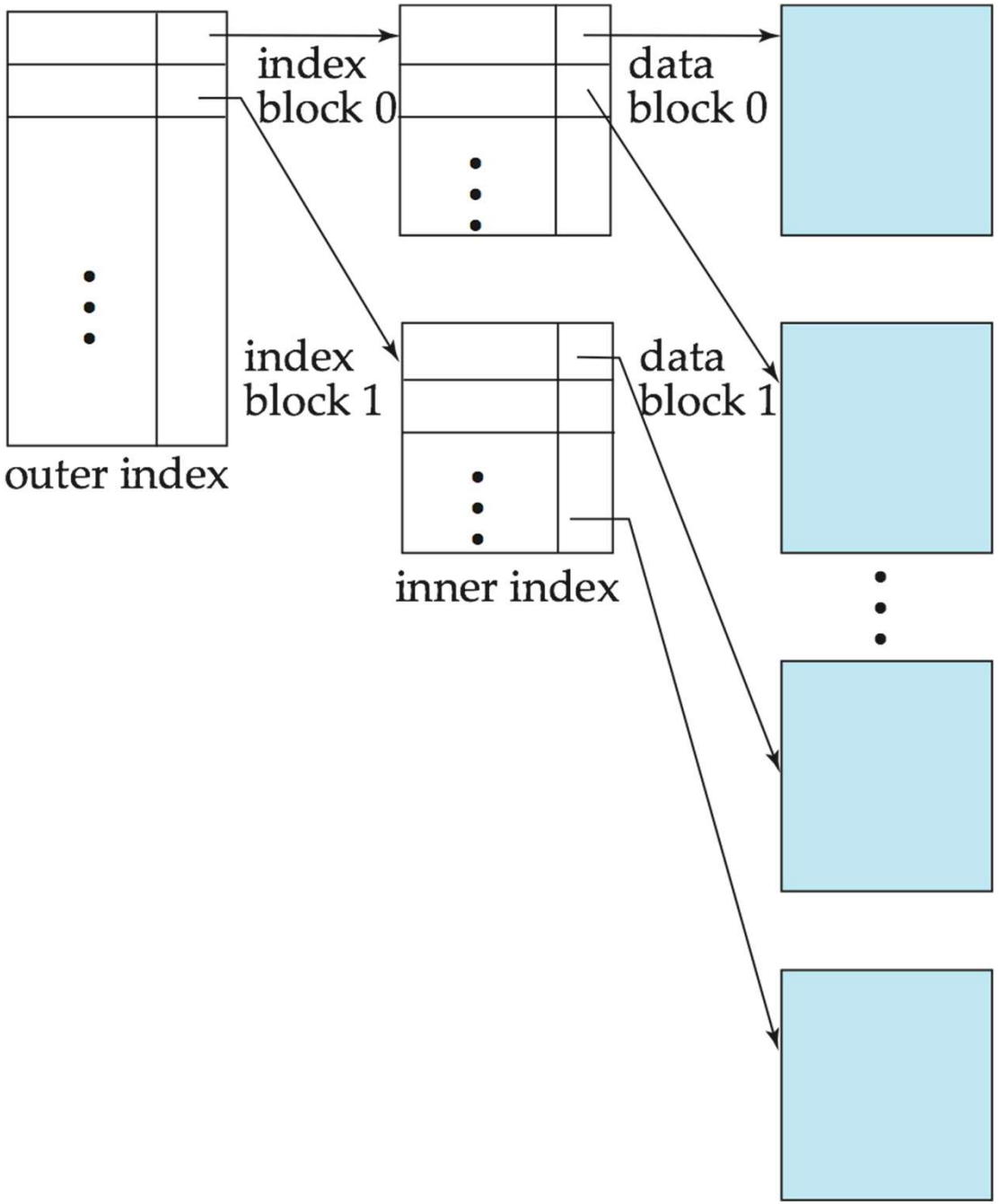
Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

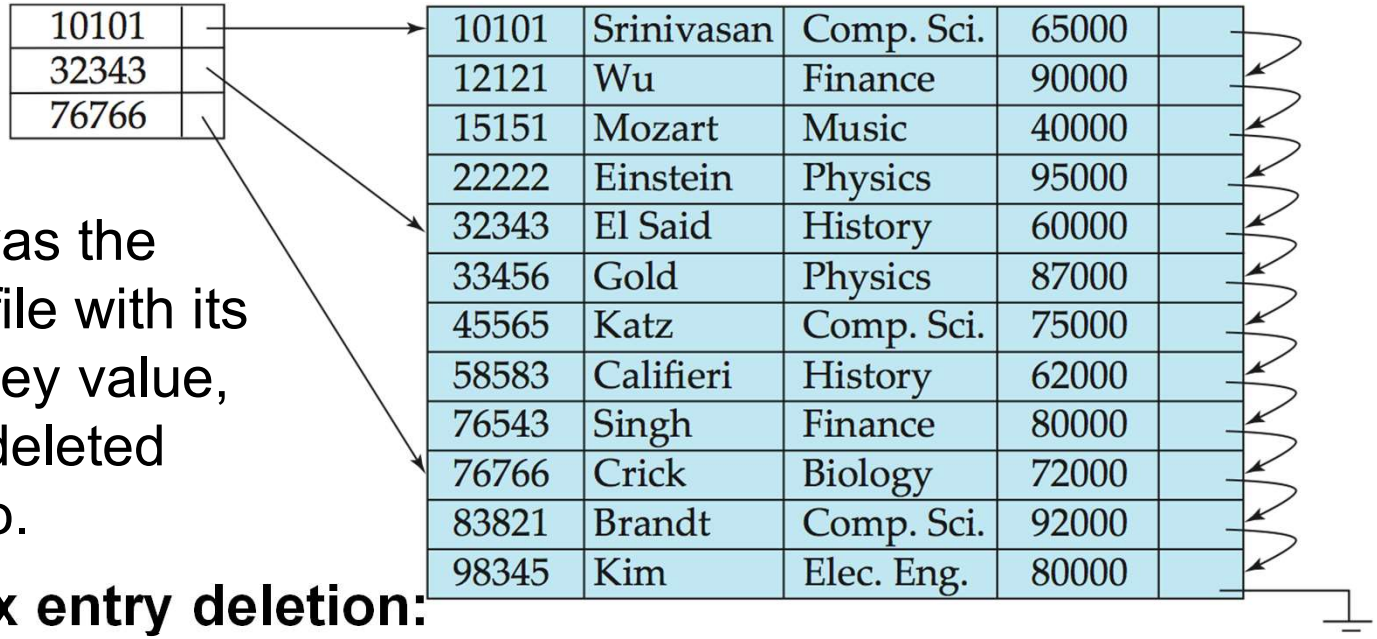
Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

Secondary Indices

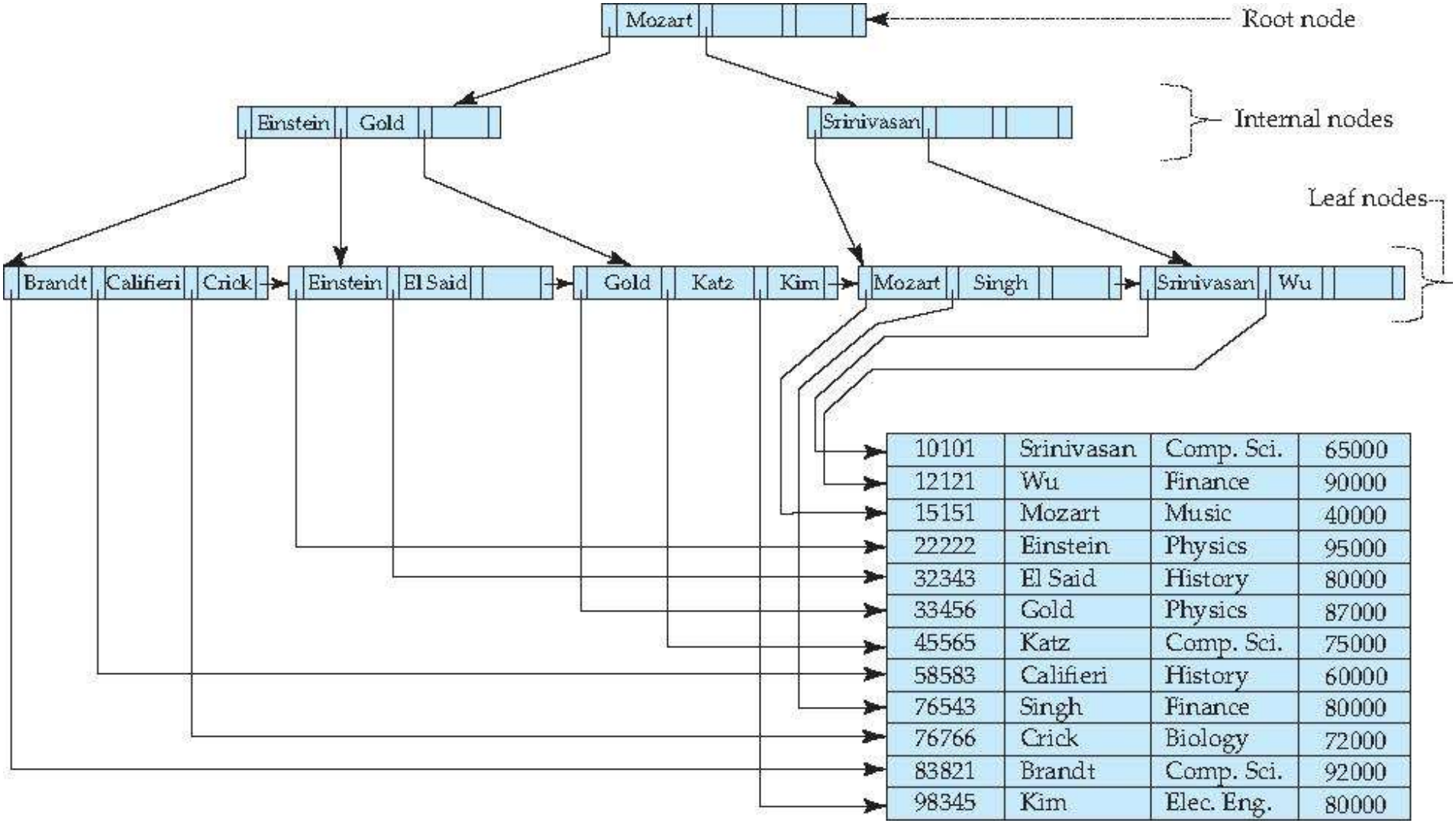
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B+-Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

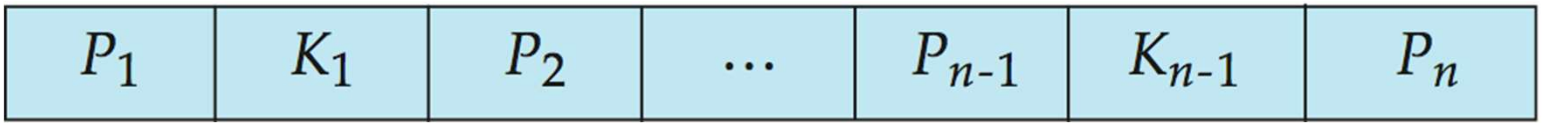
B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-Tree Node Structure

- Typical node



  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

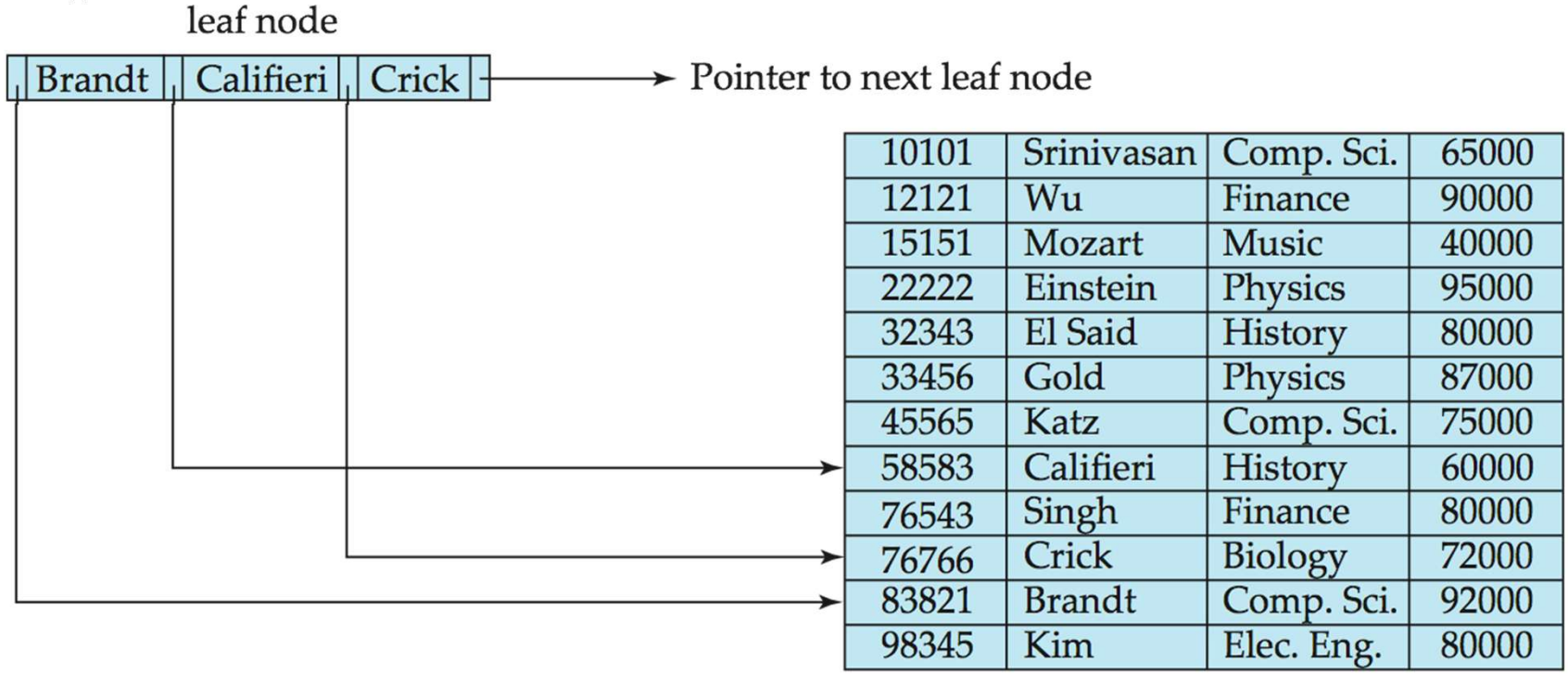
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees

Properties of a leaf node:

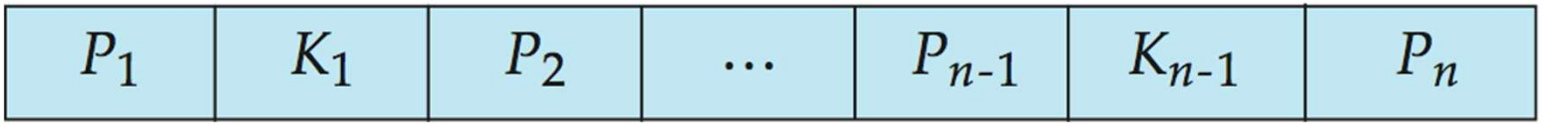
- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



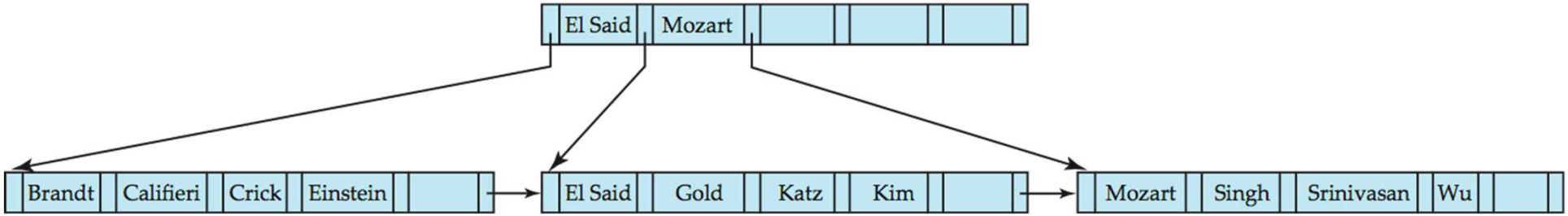
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}



Example of B+-tree



B+-tree for *instructor* file (n = 6)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with n = 6).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with n =6).
- Root must have at least 2 children.

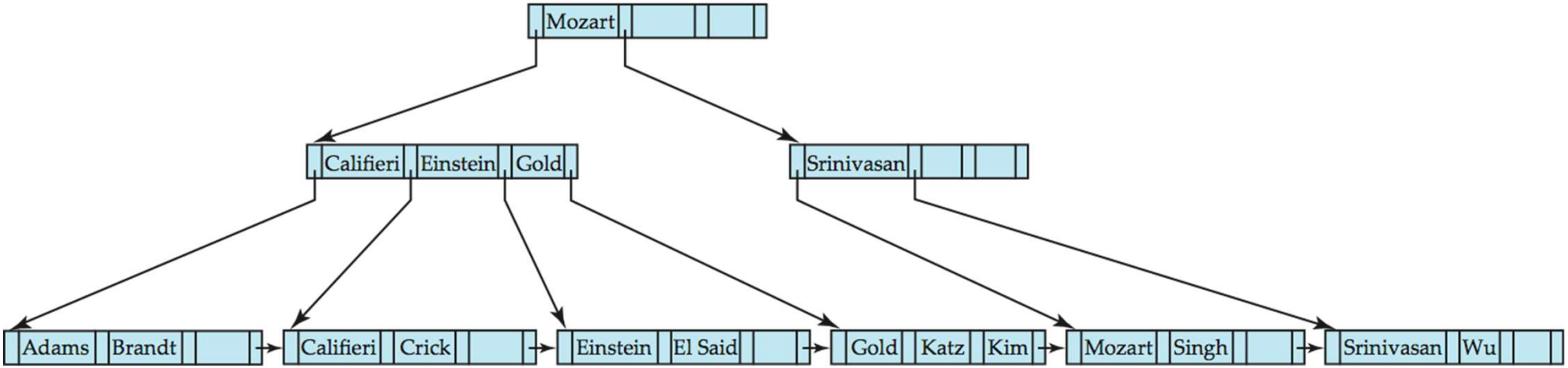
Observations about B+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
  - Level below root has at least $2* \lceil n/2 \rceil$ values
  - Next level has at least $2* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on B+-Trees

- Find record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value s.t. $V \leq K_i$.
     2. If no such exists, set *C* = *last non-null pointer in C*
     3. Else { if (V= K_i) Set C = P_{i+1} else set *C* = P_i}

     }
  3. Let *i* be least value s.t. $K_i = V$
  4. If there is such a value *i,* follow pointer P_i to the desired record.
  5. Else no record with search-key value *k* exists.



Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
  - Search-keys in the subtree to which P_i points
    - are $\leq K_{i,}$, but not necessarily $< K_{i,}$
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V

Handling Duplicates

- We modify find procedure as follows
  - traverse P_i even if $V = K_i$
  - As soon as we reach a leaf node C check if C has only search key values less than V
    - if so set C = right sibling of C before checking whether C contains V
- Procedure printAll
  - uses modified find procedure to find first occurrence of V
  - Traverse through consecutive leaves to find all occurrences of V

** Errata note: modified find procedure missing in first printing of 6th edition

Queries on B+-Trees (Cont.)

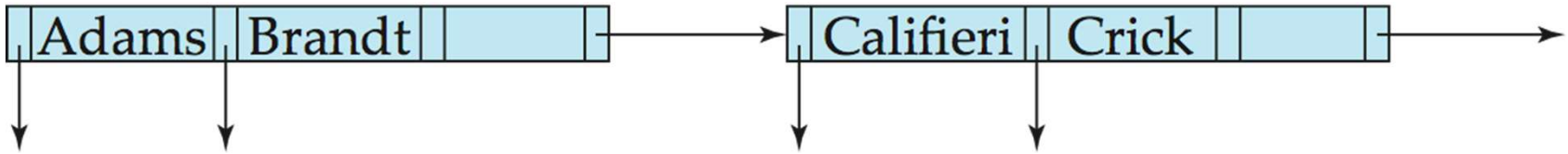
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
    2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

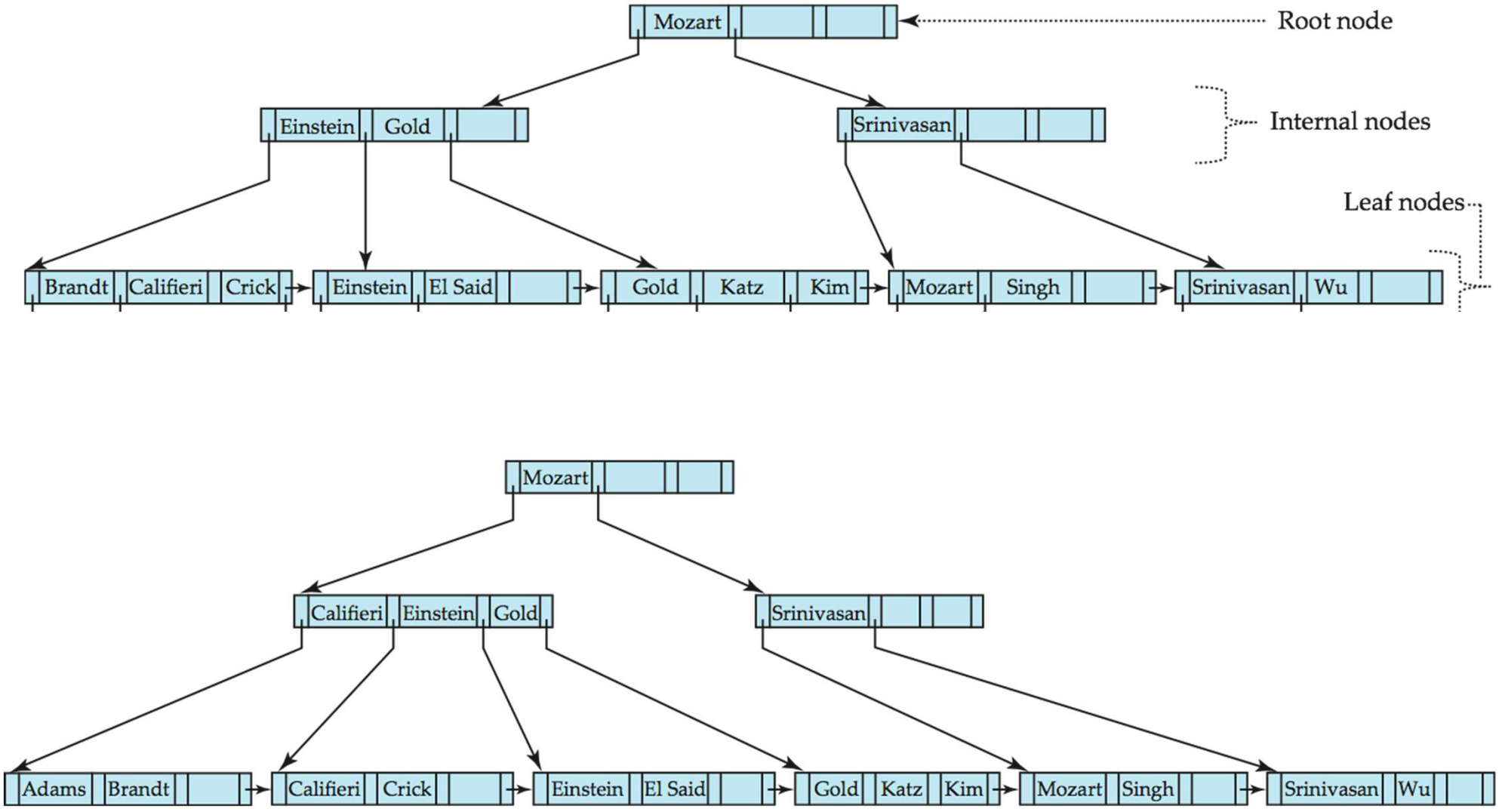
Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.



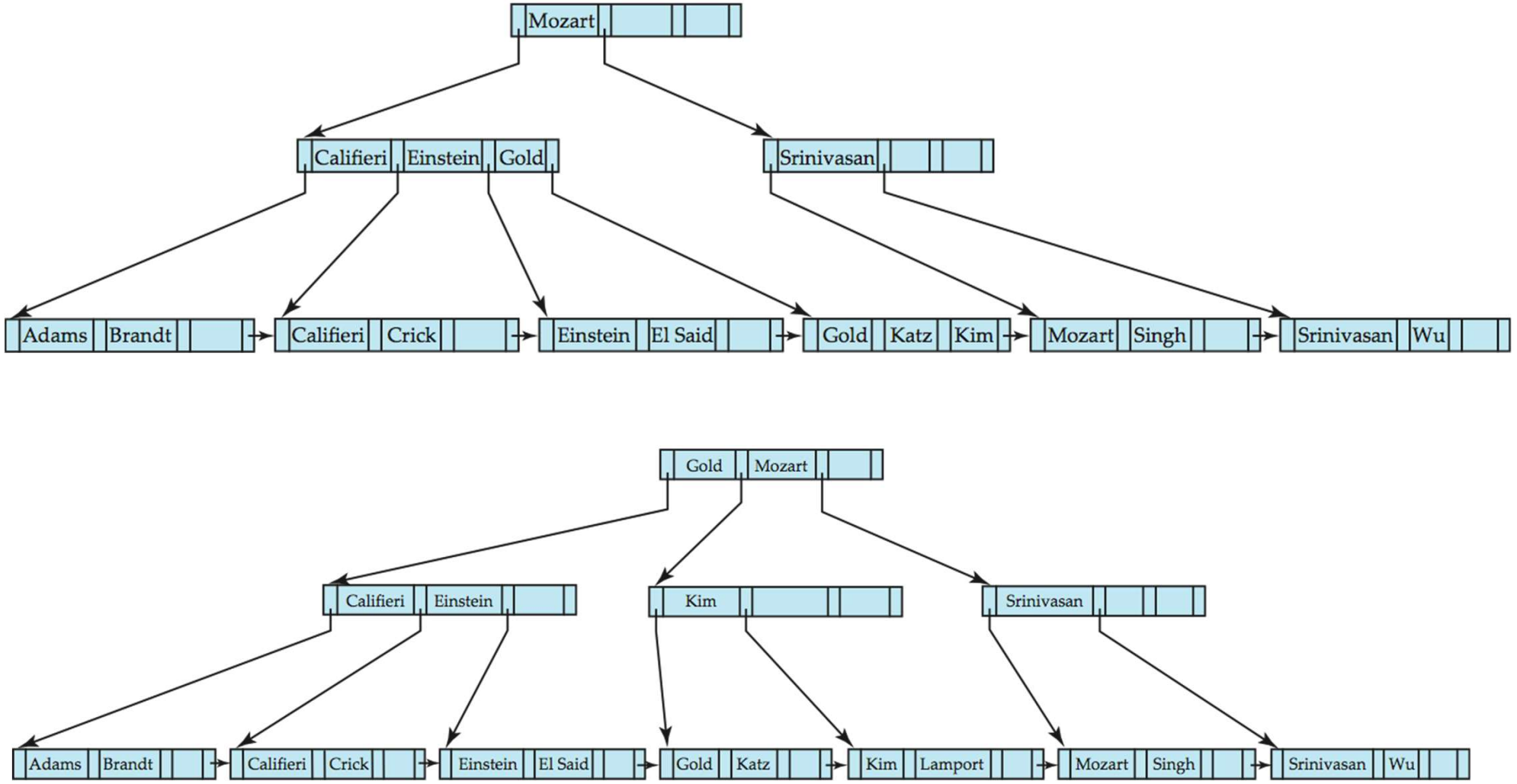
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion



B+-Tree before and after insertion of “Adams”

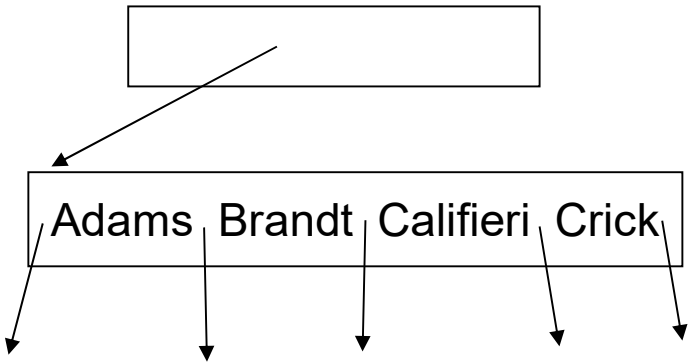
B+-Tree Insertion

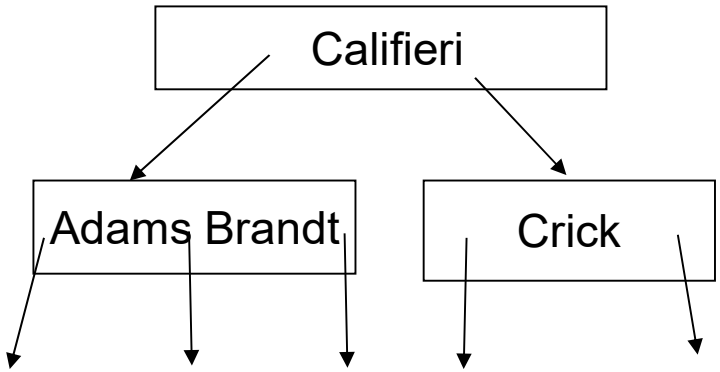


B+-Tree before and after insertion of “Lamport”

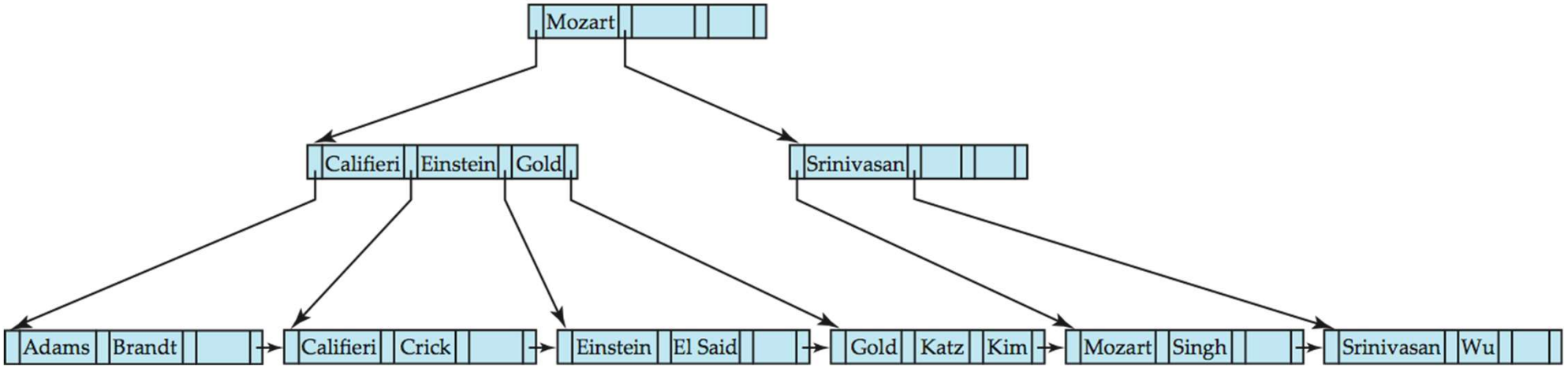
Insertion in B+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- **Read pseudocode in book!**

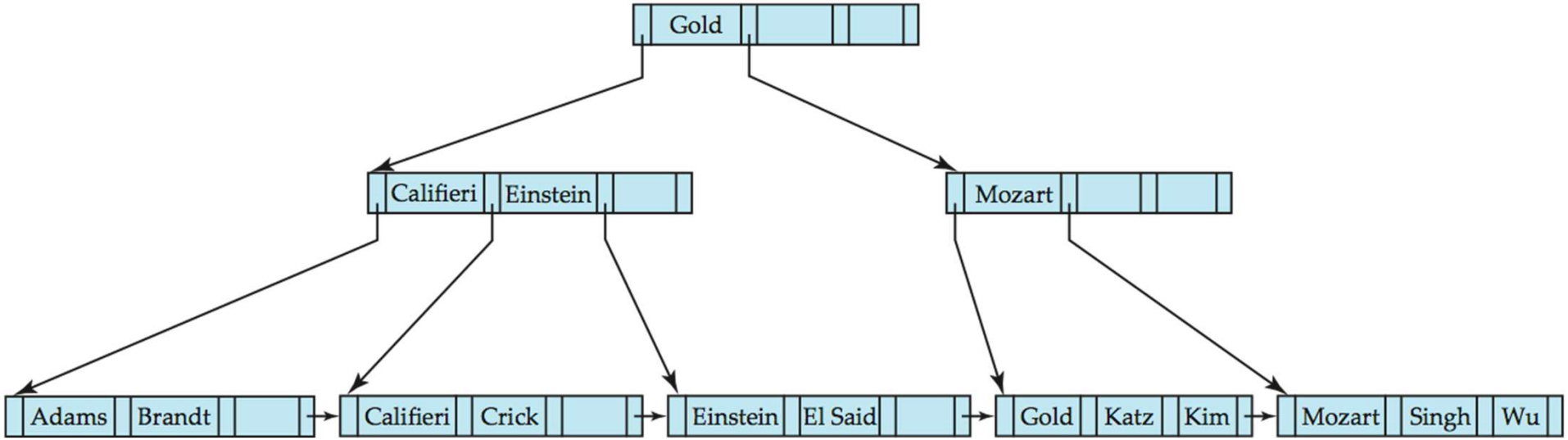




Examples of B+-Tree Deletion

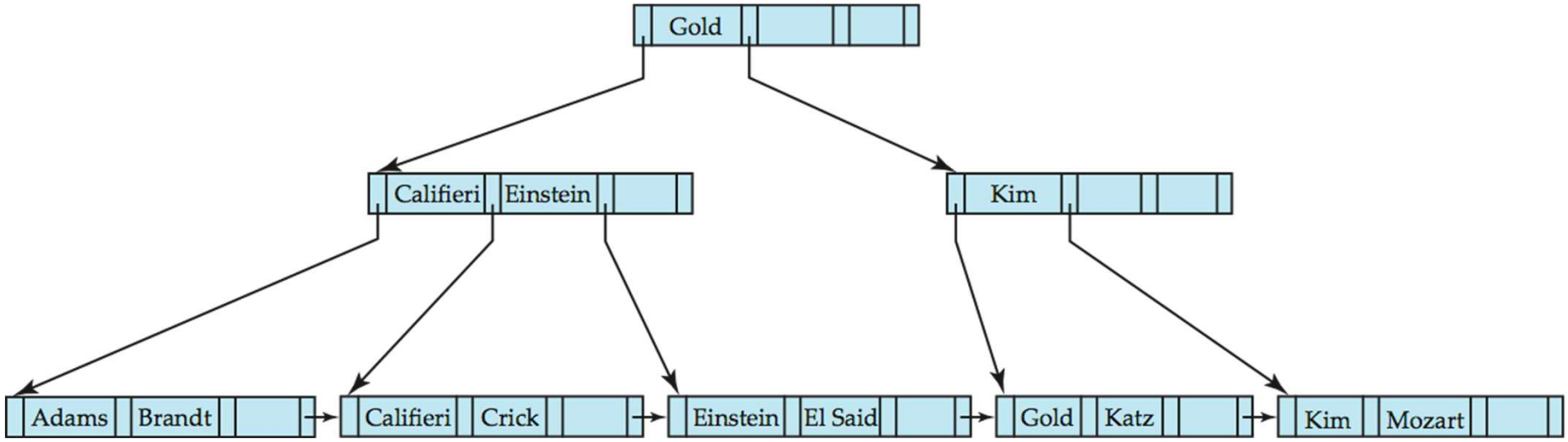


Before and after deleting “Srinivasan”



- Deleting “Srinivasan” causes merging of under-full leaves

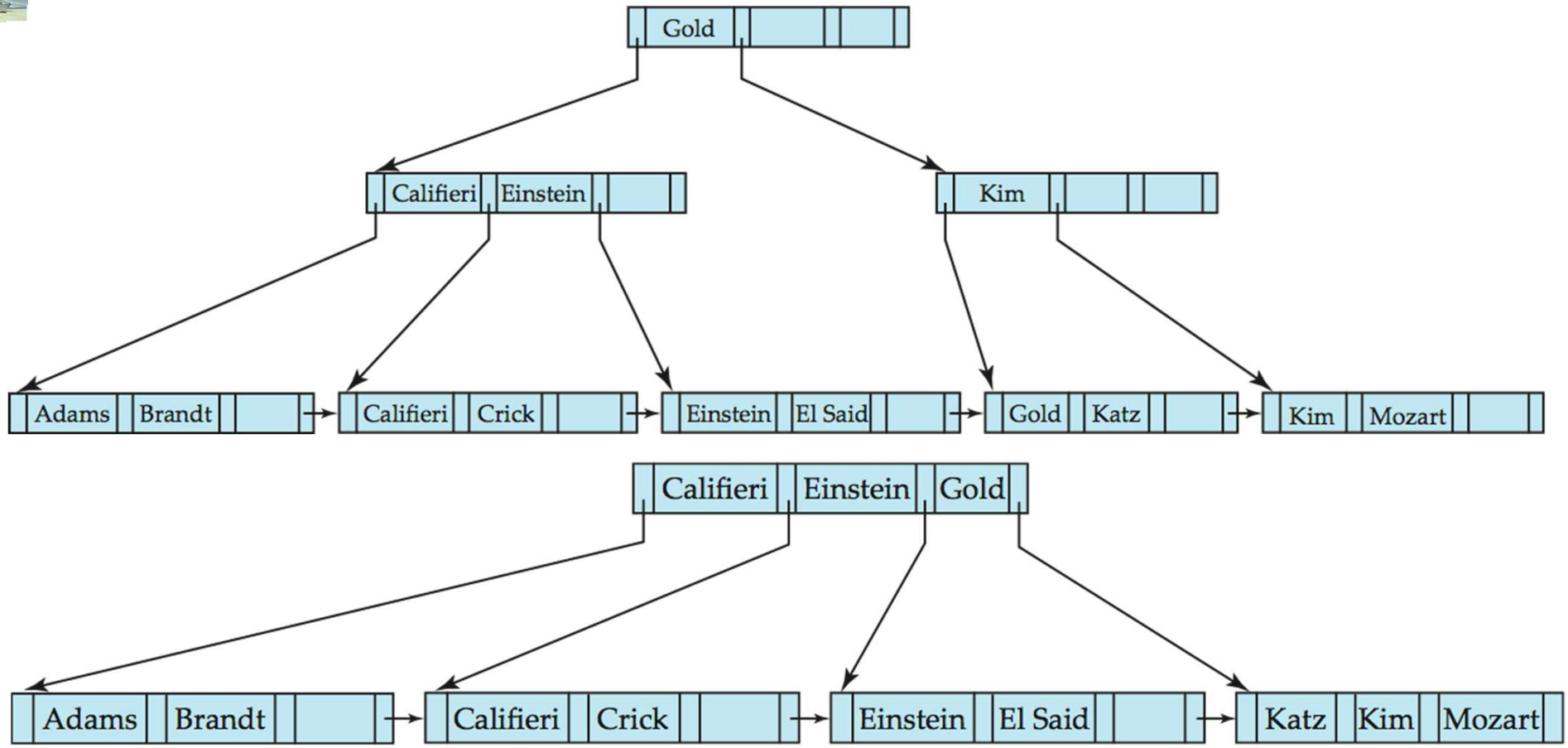
Examples of B+-Tree Deletion (Cont.)



Deletion of “Singh” and “Wu” from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

Example of B+-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Updates on B+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

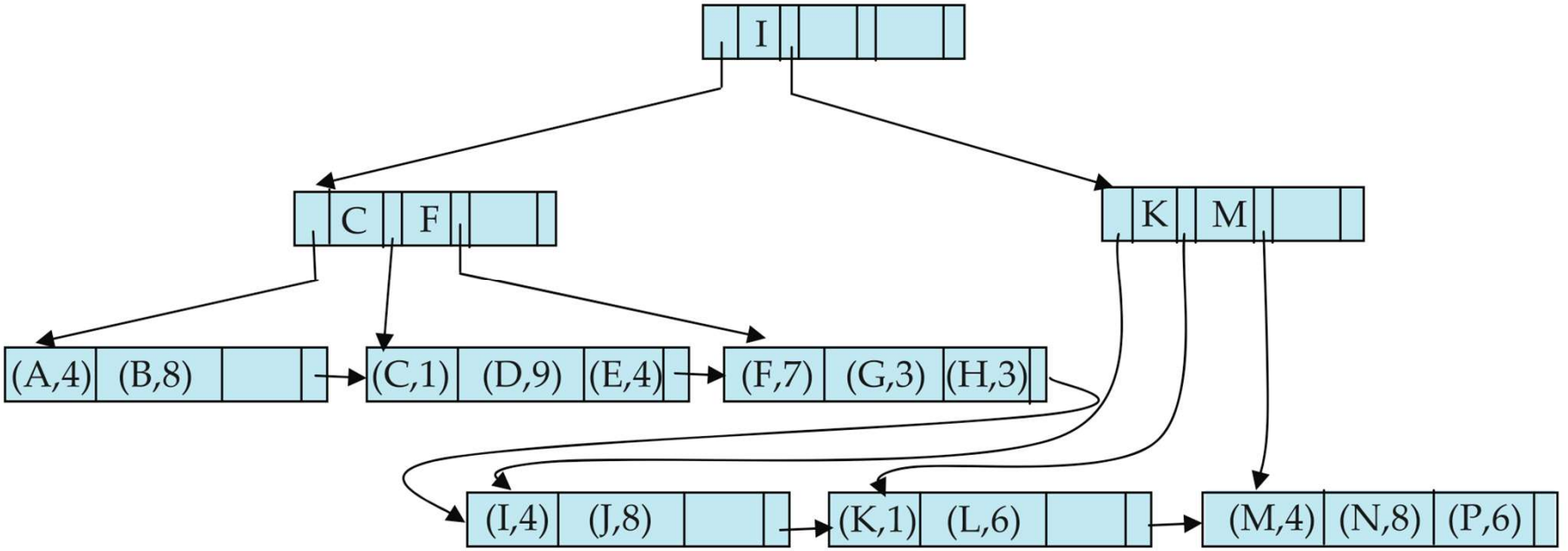
Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices.
- Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)



Example of B+-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Other Issues in Indexing

- **Record relocation and secondary indices**
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B^+-tree file organizations become very expensive
  - *Solution*: use primary-index search key instead of record pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g. “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys in leaf node can be compressed by sharing common prefixes

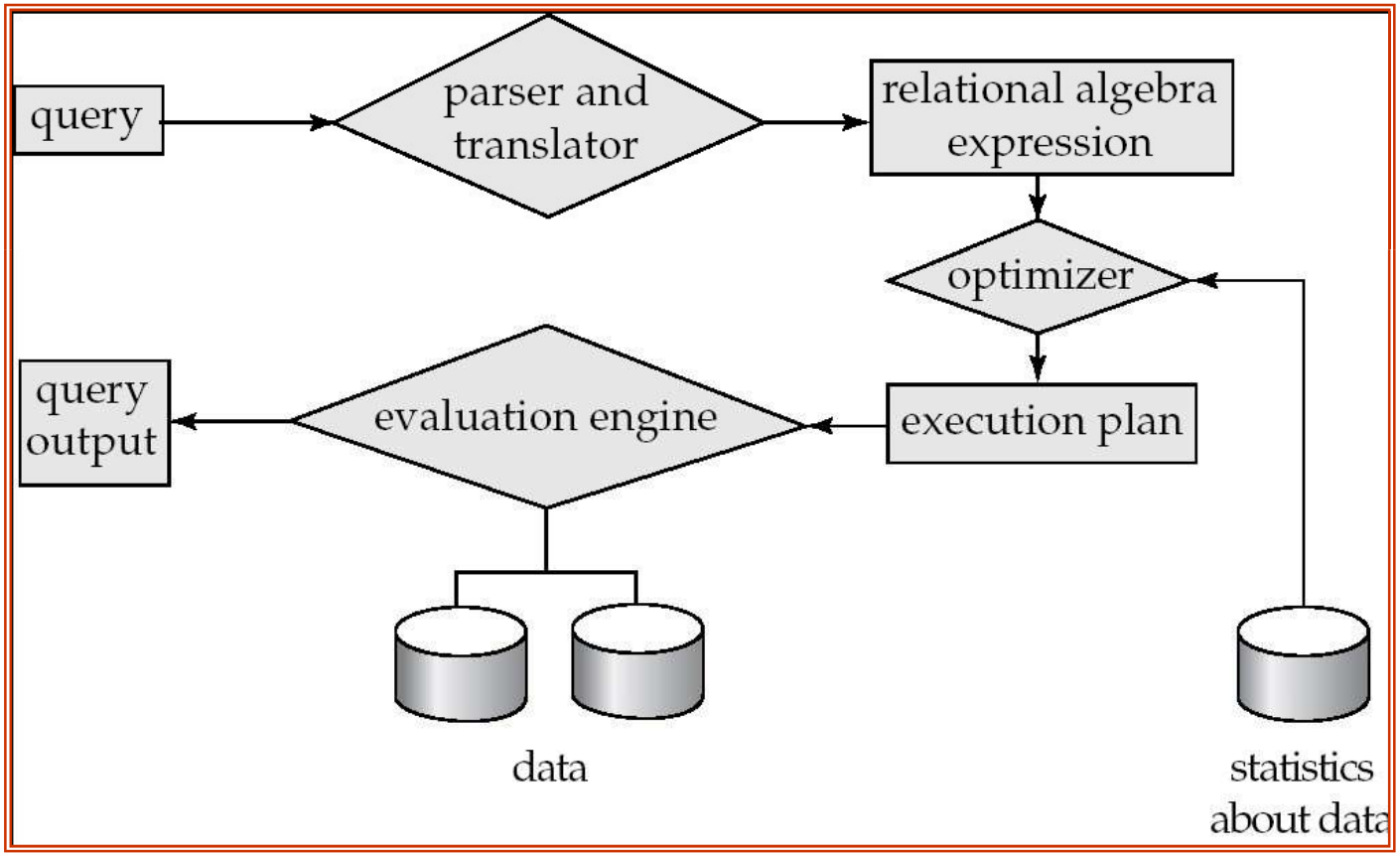
Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires $\geq$ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

LU41 - Query Processing - Measures of Query Cost

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Parsing and Evaluation

- Parsing and translation
  - Translate the query into its internal form.
  - This is then translated into relational algebra.
    - (Extended) relational algebra is more compact, and differentiates clearly among the various different operation
  - Parser checks syntax, verifies relations
  - This is a subject for *compilers*
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
    - The bulk of the problem lies in how to come up with good evaluation plans!
    - This is “simply” executing a predefined plan (or program)

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to $\Pi_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or perform complete relation scan and discard accounts with balance ≥ 2500

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
- We start by showing
  - Measure query costs (to have a measure on which to evaluate the various algorithms and plans afterwards)
  - Algorithms for evaluating (main) relational algebra operations
  - Combine algorithms for individual operations in order to evaluate a complete expression
  - How these algorithms and combinations can be parallelized, in case parallel machines are available
- We continue then to study how to optimize queries
  - That is, how to find an evaluation plan with lowest estimated cost

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
- Many factors contribute to time cost
  - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - The cost to write a block is greater than the cost to read a block
      - data is read back after being written to ensure that the write was successful
    - The cost of a seek is usually much higher than that of a block transfer read or write (one order of magnitude)

Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

$b * t_T + S * t_S$

- We do not include cost to writing output to disk in the cost formulae
- We ignore CPU costs for simplicity, as they tend to be much lower
  - Real systems do take CPU cost into account, but they are clearly less significant

- The measures of query evaluation in terms of block transfers and seeks is substantially different from that in term of number of steps

LU42 - Selection & Sorting

Selection Operation (recall)

- Notation: $\sigma_p(r)$
- p is the selection predicate
- Defined by:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

in which p is a formula of propositional calculus of terms connected by: $\wedge$ (and), $\vee$ (**or**), $\neg$ (**not**)

Each term is of the form:

<attribute> *op* <attribute> or <constant> where *op* can be one of: =, $\neq$, >, $\geq$. <. $\leq$

- Selection example:

$\sigma_{branch\text{-}name=\text{'Perryridge'}}(account)$

Algorithms for Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation *r*
  - If selection is on a key attribute, can stop on finding record(key attribute will not repeat-unique)
    - cost = (b_r /2) block transfers + 1 seek
  - This linear search can be always applied, regardless of:
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Select * from emp where ename like ‘k%’;

Cost: 18 blocks+ 1 seek (tb- 3 sec+ts-2sec) 18*3+1*2= 56 sec take to finish task Select * from emp where eno=2;

Cost: 9 blocks+1 seek 9*3+1*2=29 sec

Algorithms for Selection (Cont.)

- **A2** *(binary search).* Applicable only if the selection is an equality comparison on the attribute on which file is ordered.
  - Assumes that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost later
  - If b_r is not too big, then most likely binary search doesn't pay.
    - Note that t_T is several (say, 50) times bigger than t_S
    - Estimates on the size of the relation are needed to wisely choose which of the two algorithms is better for a specific query at hands.

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$

where h_i denotes the height of the index

- Recall that the height of a B+-tree index is $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$, where n is the number of index entries per node and K is the number of search keys.
  - E.g. for a relation r with 1.000.000 different search key, and with 100 index entries per node, $h_i = 4$
  - Unless the relation is really small, this algorithms always “pays” when indexes are available

Selections Using Indices (cont)

- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = $(h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive if n is big!
        - Note that it multiplies the time for seeks by *n*.

Selections Involving Comparisons

- One can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan or binary search just as before,
  - or by using indices in the following ways:
- **A6** (*primary index, comparison).*
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > *v;*
  - Using the index would be useless, and would requires extra seeks on the index file
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record (a lot!)
    - Linear file scan may be much cheaper!!!!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1}\sigma_{\theta_2}\ldots\sigma_{\theta_n}(r)$
- **A8** (*conjunctive selection using one index).*
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
  - In this case the choice of the first condition is crucial!
- One must use estimates to know which one is better.
- **A9** (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- **A10** (*conjunctive selection by intersection of identifiers).*
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1} \vee_{\theta_2} \vee \ldots \vee_{\theta_n} (r)$.
- **A11** (*disjunctive selection by union of identifiers*).
  - Applicable only if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Sorting

- Sorting algorithms are important in query processing at least for two reasons:
  - The query itself may require sorting (**order by** clause)
  - Some algorithms for other operations, like join, set operations and aggregation, require that relation are previously sorted

- To sort a relation:
  - We may build an index on the relation, and then use the index to read the relation in sorted order.
    - This only sorts the relation logically, not physically
    - May lead to one disk block access for each tuple.
  - For relations that fit in memory sorting algorithms that you've studied before, like quicksort, can be used.
    - For relations that don't fit in memory special algorithms are required, that take into account the measures in terms of disc transfers and seeks.

External Sort-Merge

- It is a sorting algorithm to be used when the whole relation does not fit in memory.

Let M denote memory size (in pages).

1. **Create sorted runs.** Let *i* be 0 initially.

Repeatedly do the following till the end of the relation:

(a) Read *M* blocks of relation into memory
(b) Sort the in-memory blocks (with your favorite sorting algorithm)
(c) Write sorted data to run R_i; increment *i.*

Let the final value of *i* be *N*

2. *Merge the runs (next slide)…..*

External Sort-Merge (Cont.)

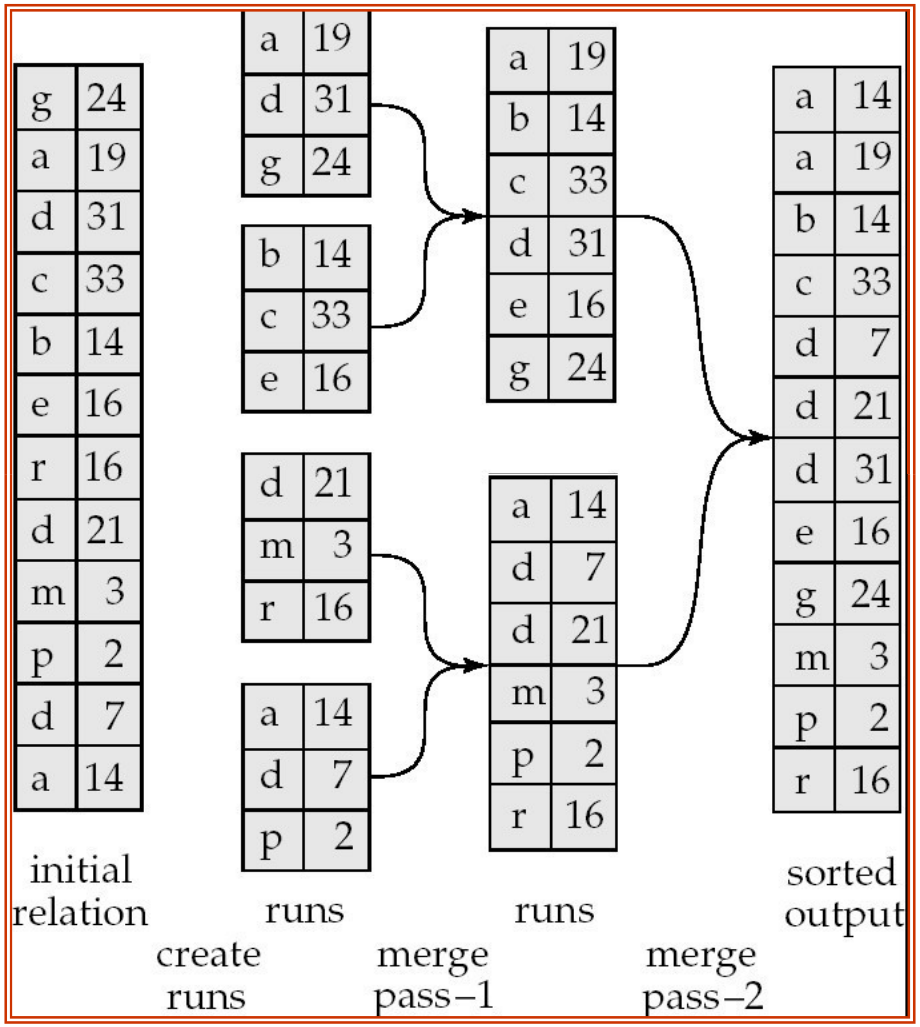
2. **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.
    1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
    2. **repeat**
        1. Select the first record (in sort order) among all buffer pages
        2. Write the record to the output buffer. If the output buffer is full write it to disk.
        3. Delete the record from its input buffer page.

    If the buffer page becomes empty **then**
    read the next block (if any) of the run into the buffer.
    3. **until** all input buffer pages are empty:

External Sort-Merge (Cont.)

- If N M, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
- E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

Example: External Sorting Using Sort-Merge



External Merge Sort Transfer Cost

- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost

– we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

  - Thus total number of block transfers for external sorting:

$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$

External Merge Sort Seek Cost

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Buffer size: b_b (read/write b_b blocks at a time)
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass

– except the final one which does not require a write

- Total number of seeks:

$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$

LU43 – Join Operations

Join Operation

- Several different algorithms to implement joins exist (not counting with the ones involving parallelism)
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- As for selection, the choice is based on cost estimate

- Examples in next slides use the following information
  - Number of records of *customer*: 10.000 *depositor*: 5.000
  - Number of blocks of *customer*: 400 *depositor*: 100

Nested-Loop Join

- The simplest join algorithms, that can be used independently of everything (like the linear search for selection)
- To compute the theta join $r \bowtie_{\theta} s$

for each tuple t_r **in** r **do begin**
for each tuple t_s **in** s **do begin**
test pair (t_r, t_s) to see if they satisfy the join condition θ
if they do, add $t_r \bullet t_s$ to the result.
end end

- r is called the **outer relation** and s the **inner relation** of the join.

- Quite expensive in general, since it requires to examine every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$n_r \times b_s + b_r$

block transfers, plus

$n_r + b_r$

seeks

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to b_r + b_s block transfers and 2 seeks
- In general, it is much better to have the smaller relation as the outer relation
  - The number of block transfers is multiplied by the number of blocks of the inner relation
- However, if the smaller relation is small enough to fit in memory, one should use it as the inner relation!
- The choice of the inner and outer relation strongly depends on the estimate of the size of each relation

Nested-Loop Join Cost in Example

- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5.000 □ 400 + 100 = 2.000.100 block transfers,
    - 5.000 + 100 = 5100 seeks
  - with *customer* as the outer relation
    - 10.000 □ 100 + 400 = 1.000.400 block transfers, 10.400 seeks
- If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks

- Instead of iterating over records, one could iterate over blocks. This way, instead of n_r □ b_s + b_r we would have b_r □ b_s + b_r block transfers
- This is the basis of the block nested-loops algorithm (next slide).

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

for each block B_r **of** r **do begin**
for each block B_s **of** s **do begin** **for each** tuple t_r **in** B_r **do begin**
for each tuple t_s **in** B_s **do begin**
Check if (t_r, t_s) satisfy the join condition if they do, add $t_r \bullet t_s$ to the result.
end **end**
end **end**

Block Nested-Loop Join Cost

- Worst case estimate: $b_r \cdot b_s + b_r$ block transfers and 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation
- Best case (when smaller relation fits into memory): $b_r + b_s$ block transfers plus 2 seeks.
- Some improvements to nested loop and block nested loop algorithms can be made:
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available to faster get the tuples which match the tuple of the outer relation at hands

Indexed Nested-Loop Join

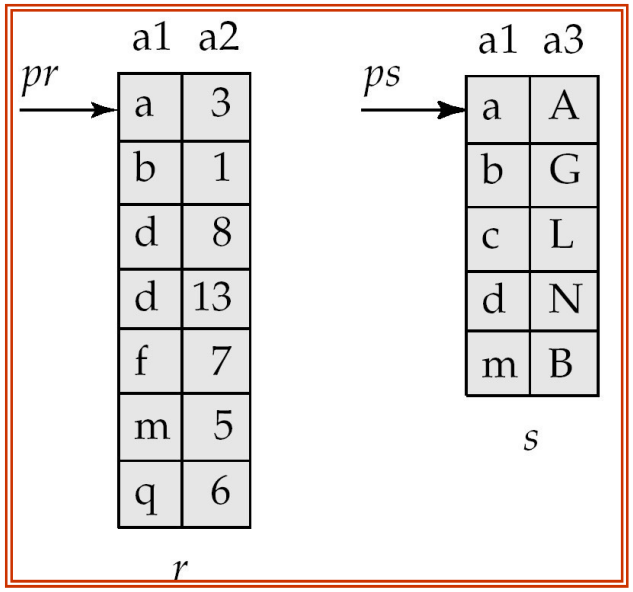
- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
- In some cases, it pays to construct an index just to compute a join.
- For each tuple t_r in the outer relation r, use the index on s to look up tuples in s that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of r, and, for each tuple in r, we perform an index lookup on s.
- Cost of the join: $b_r (t_T + t_S) + n_r \square c$
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple or r
  - c can be estimated as cost of a single selection on s using the join condition (usually quite low, when compared to the join)
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *depositor* ⋈ *customer,* with *depositor* as the outer relation.
  - Let *customer* have a primary B+-tree index on the join attribute *customer-name,* which contains 20 entries in each index node.
- Since *customer* has 10.000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5.000 tuples
- For nested loop: 2.000.100 block transfers and 5.100 seeks
- Cost of block nested loops join
- 400*100 + 100 = 40.100 block transfers + 2 * 100 = 200 seeks
- Cost of indexed nested loops join
  - 100 + 5.000 * 5 = 25.100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join
  - However in terms of time for transfers and seeks, in this case using the index doesn't pay (this is specially so because the relations are small)

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
    1. Join step is similar to the merge stage of the sort-merge algorithm.
    2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
    3. Detailed algorithm may be found in the book



Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming that all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is (where b_b is the number of blocks in allocated in memory):

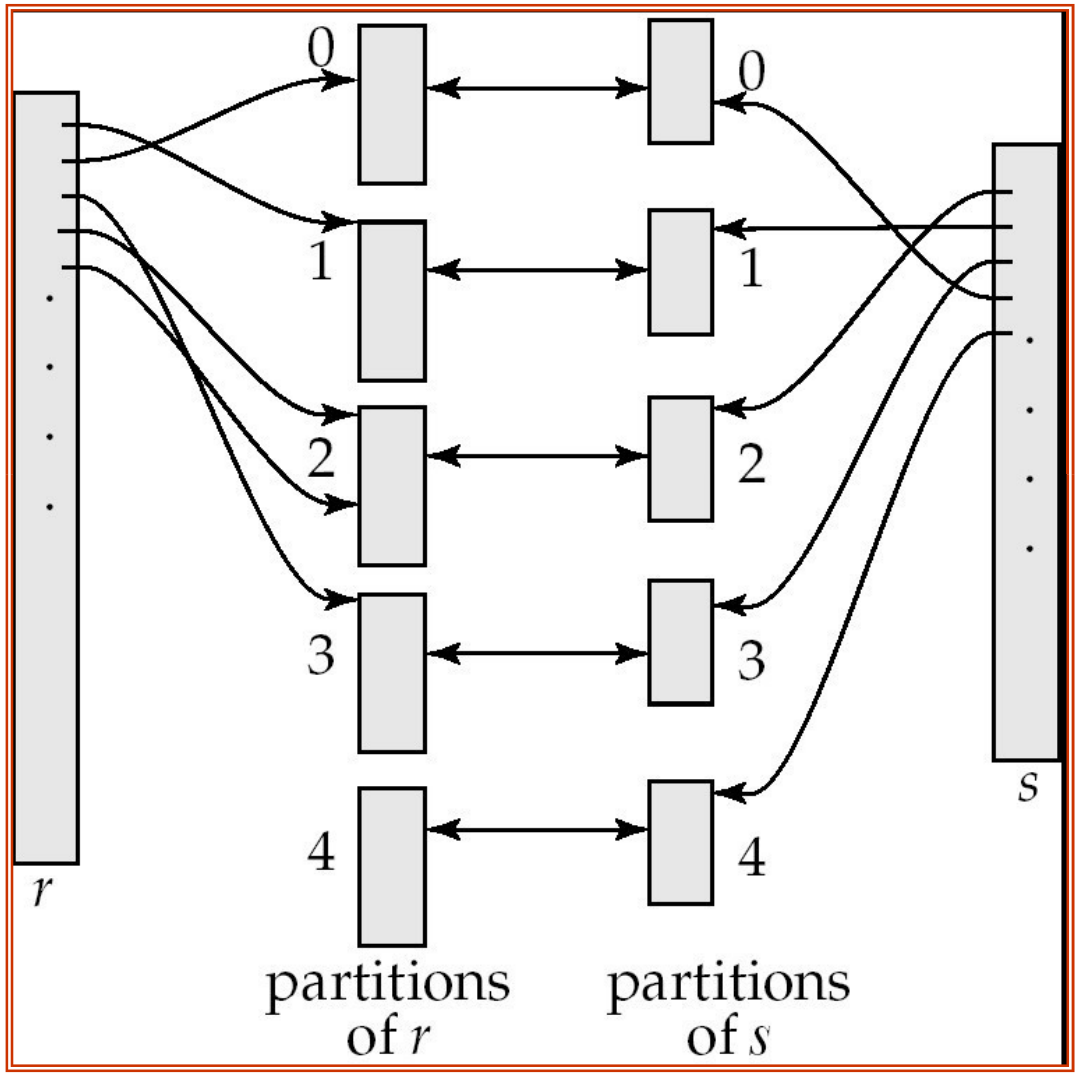
$b_r + b_s$ block transfers + $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

- + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B$^+$-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
- Sequential scan more efficient than random lookup

Hash-Join

- Also only applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- General idea:
  - Partition the relations according to this
  - Then perform the join on each partition r_i and s_i
    - There is no need to compute the join between different partitions since an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes. If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join (Cont.)



Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.

2. Partition *r* similarly.

3. For each *i:*

   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a ***different hash*** function than the earlier one *h.*

   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and
r is called the **probe input.**

Hash-Join algorithm (Cont.)

- The number of partitions *n* for the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2, to avoid overflows
    - The probe relation partitions r_i need not fit in memory
    - **Recursive partitioning** required if number of partitions *n* is greater than number of pages *M* of memory.
      - instead of partitioning *n* ways, use *M* – 1 partitions for s
  - Further partition the *M* – 1 partitions using a different hash function
  - Use same partitioning method on *r*
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.
    - So is not further considered here (see the book for details on the associated costs)

Cost of Hash-Join

- The cost of hash join is
$3(b_r + b_s) + 4 \lceil n_h \rceil$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If the entire build input can be kept in main memory no partitioning is required
- Cost estimate goes down to $b_r + b_s$.
- For the running example, assume that memory size is 20 blocks
- $b_{depositor}$ = 100 and $b_{customer}$ = 400.
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *customer* into five partitions, each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
- 3(100 + 400) = 1.500 block transfers + $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil)$ = 336 seeks
- The best we had up to here was 40.100 block transfers plus 200 seeks (for block nested loop) or 25,100 block transfers and seeks (for index nested loop)

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

  - Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s)$$

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection:**
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in similarly to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimization:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
- When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

- **Set operations** (□, □ and □): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - *r* □ *s*:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i add the tuples in the hash index to the result.
        - *r* □ *s*:
          1. output tuples in s_i to the result if they are already there in the hash index
        - *r* – *s:*
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.
- Modifying merge join to compute $r \mathbin{⟕} s$
  - In $r \mathbin{⟕} s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r \mathbin{⟕} s$: During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.
- Modifying hash join to compute $r \mathbin{⟕} s$
  - If r is probe relation, output non-matching r tuples added with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

LU44 - Evaluation of Expressions

Evaluation of Expressions

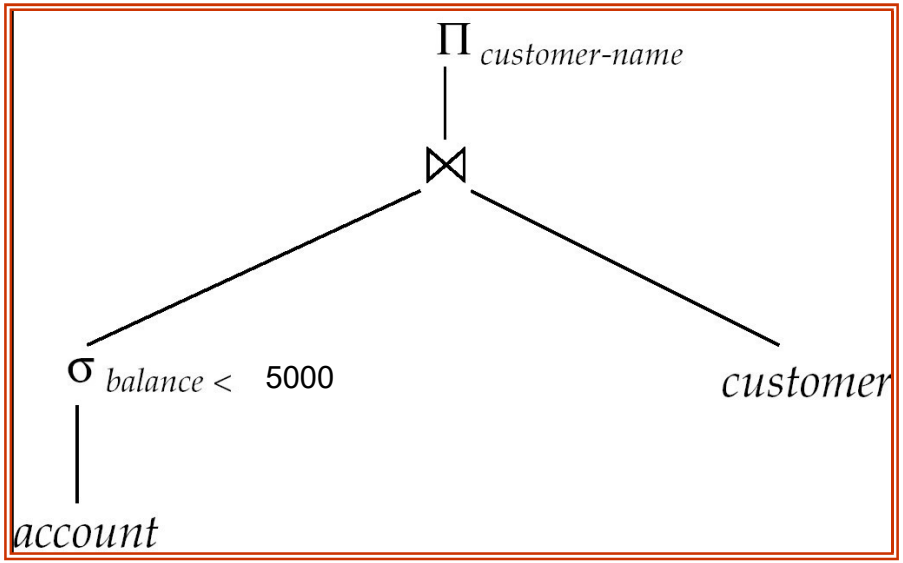
- So far we have seen algorithms for individual operations
  - These have then to be combined to evaluate complex expressions, with several operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{balance<5000}(account)$$

then compute the store its join with *customer,* and finally compute the projections on *customer-name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- It may require considerable storage space. Moreover, cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so

Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

- Double buffering: use two output buffers for each operation, when one is full, write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.
  - Similarly, don't store result of join, pass tuples directly to projection.
- It is much cheaper than materialization: there is no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort and hash-join where a preliminary phase is required over the whole relations
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **producer-driven** (or **eager** or **push**) pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
  - In **demand driven**(or **lazy**, or **pull**) evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next

Pipelining (Cont.)

- Implementation of pull pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - open()
      - E.g. file scan: initialize file scan
    - state: pointer to beginning of file
      - E.g.merge join: sort relations;
    - state: pointers to beginning of sorted relations
    - next()
      - E.g. for file scan: Output next tuple, and advance and store file pointer
      - E.g. for merge join: continue with merge from earlier state till

next output tuple is found. Save pointers as iterator state.

    - close()

Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - intermediate results written to disk and then read back
- Algorithm variants to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join (see book for details) generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
- It is clear that pipelining could greatly benefit from parallel processing, especially of sufficiently independent sub-expressions
  - And this is not the only chance for parallelism in query processing!

LU45 - Query Tuning

Query Tuning

- *Query tuning refers specifically to optimizing the performance of individual SQL queries. The goal is to make sure that each query runs as efficiently as possible, usually by reducing its execution time and resource consumption. Key activities involved in query tuning include:*
- **Optimizing the SQL syntax:** Refactoring the query to use more efficient SQL constructs (e.g., replacing subqueries with joins).
- **Indexing:** Creating or modifying indexes to speed up data retrieval.
- **Query plan analysis:** Examining the query execution plan to identify bottlenecks (e.g., full table scans or inefficient joins).
- **Using appropriate filtering:** Ensuring that WHERE clauses are used effectively to reduce the amount of data being processed.
- **Reducing joins or subqueries:** Minimizing unnecessary joins or simplifying nested queries to improve performance.

Performance Tuning

- Adjusting various parameters and design choices to improve system performance for a specific application.
- Tuning is best done by
  1. identifying bottlenecks, and
  2. eliminating them.
- Can tune a database system at 3 levels:
  - **Hardware** -- e.g., add disks to speed up I/O, add memory to increase buffer hits, move to a faster processor.
  - **Database system parameters** -- e.g., set buffer size to avoid paging of buffer, set checkpointing intervals to limit log size. System may have automatic tuning.
  - **Higher level database design**, such as the schema, indices and transactions (more later)

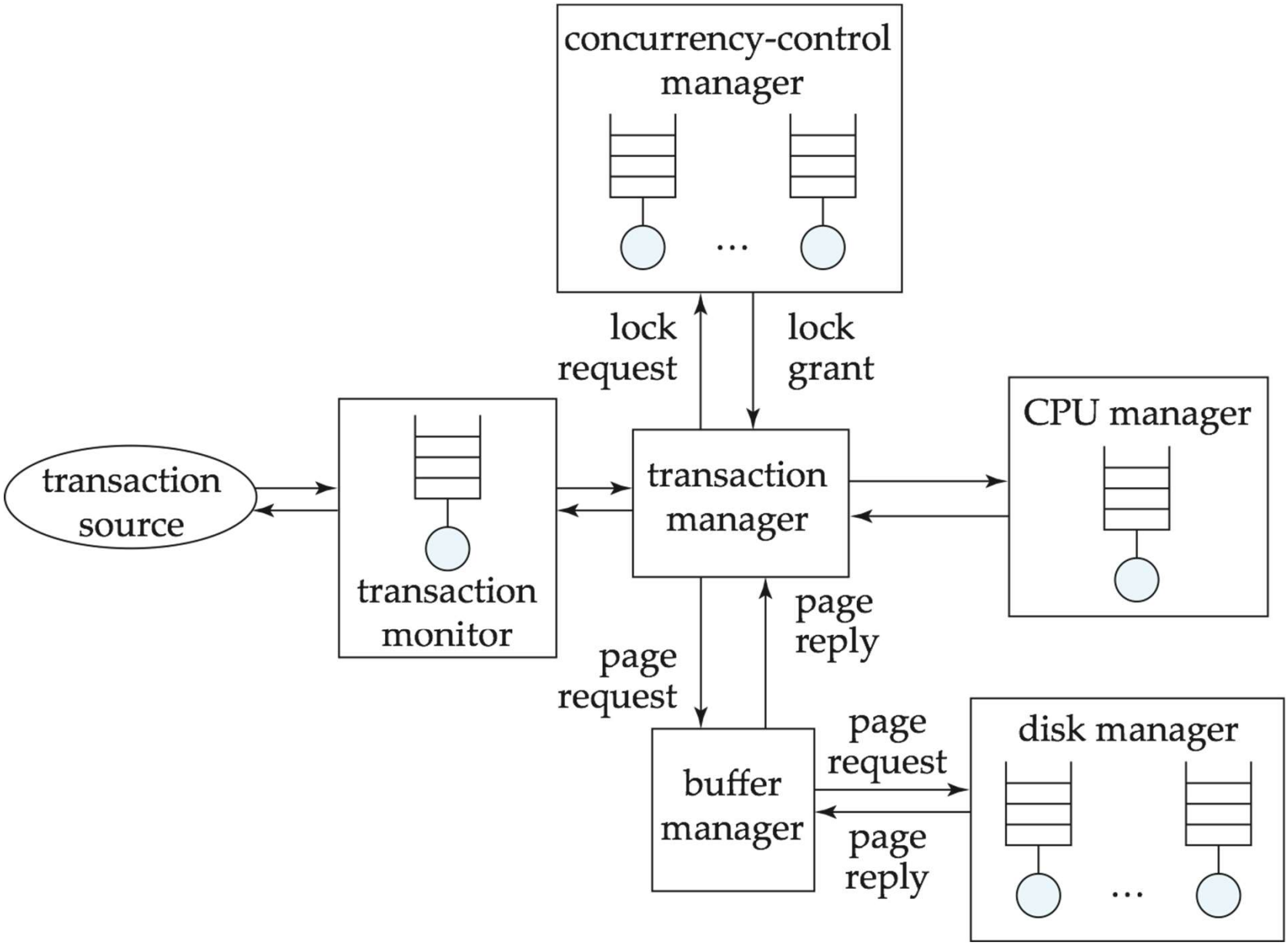
Bottlenecks

- Performance of most systems (at least before they are tuned) usually limited by performance of one or a few components: these are called **bottlenecks**
  - E.g., 80% of the code may take up 20% of time and 20% of code takes up 80% of time
    - Worth spending most time on 20% of code that take 80% of time
- Bottlenecks may be in hardware (e.g., disks are very busy, CPU is idle), or in software
- Removing one bottleneck often exposes another
- De-bottlenecking consists of repeatedly finding bottlenecks, and removing them
  - This is a heuristic

Identifying Bottlenecks

- Transactions request a sequence of services
  - E.g., CPU, Disk I/O, locks
- With concurrent transactions, transactions may have to wait for a requested service while other transactions are being served
- Can model database as a **queueing system** with a queue for each service
  - Transactions repeatedly do the following
    - request a service, wait in queue for the service, and get serviced
- Bottlenecks in a database system typically show up as very high utilizations (and correspondingly, very long queues) of a particular service
  - E.g., disk vs. CPU utilization
  - 100% utilization leads to very long waiting time:
    - Rule of thumb: design system for about 70% utilization at peak load
    - utilization over 90% should be avoided

Queues In A Database System



Tunable Parameters

- Tuning of hardware
- Tuning of schema
- Tuning of indices
- Tuning of materialized views
- Tuning of transactions

Tuning of Hardware

- Even well-tuned transactions typically require a few I/O operations
  - Typical disk supports about 100 random I/O operations per second
  - Suppose each transaction requires just 2 random I/O operations. Then to support *n* transactions per second, we need to stripe data across *n*/50 disks (ignoring skew)
- Number of I/O operations per transaction can be reduced by keeping more data in memory
  - If all data is in memory, I/O needed only for writes
  - Keeping frequently used data in memory reduces disk accesses, reducing number of disks required, but has a memory cost

Hardware Tuning: Five-Minute Rule

- Question: which data to keep in memory:
  - If a page is accessed *n* times per second, keeping it in memory saves
    - $n * \frac{\text{price-per-disk-drive}}{\text{accesses-per-second-per-disk}}$
  - Cost of keeping page in memory
    - $\frac{\text{price-per-MB-of-memory}}{\text{ages-per-MB-of-memory}}$
  - Break-even point: value of *n* for which above costs are equal
    - If accesses are more then saving is greater than cost
  - Solving above equation with current disk and memory prices leads to: **5-minute rule: if a page that is randomly accessed is used more frequently than once in 5 minutes it should be kept in memory**
    - (by buying sufficient memory!)

Hardware Tuning: One-Minute Rule

- For sequentially accessed data, more pages can be read per second. Assuming sequential reads of 1MB of data at a time:
  1-minute rule: sequentially accessed data that is accessed once or more in a minute should be kept in memory
- Prices of disk and memory have changed greatly over the years, but the ratios have not changed much
  - So rules remain as 5 minute and 1 minute rules, not 1 hour or 1 second rules!

Hardware Tuning: Choice of RAID Level

- To use RAID 1 or RAID 5?
  - Depends on ratio of reads and writes
    - RAID 5 requires 2 block reads and 2 block writes to write out one data block
- If an application requires *r* reads and *w* writes per second
  - RAID 1 requires $r + 2w$ I/O operations per second
  - RAID 5 requires: $r + 4w$ I/O operations per second
- For reasonably large r and w, this requires lots of disks to handle workload
  - RAID 5 may require more disks than RAID 1 to handle load!
  - Apparent saving of number of disks by RAID 5 (by using parity, as opposed to the mirroring done by RAID 1) may be illusory!
- Thumb rule: RAID 5 is fine when writes are rare and data is very large, but RAID 1 is preferable otherwise
  - If you need more disks to handle I/O load, just mirror them since disk capacities these days are enormous!

Tuning the Database Design

- **Schema tuning**
  - Vertically partition relations to isolate the data that is accessed most often -- only fetch needed information.
    - E.g., split *account* into two, (*account-number*, *branch-name*) and (*account-number*, *balance*).
      - Branch-name need not be fetched unless required
  - Improve performance by storing a **denormalized relation**
    - E.g., store join of *account* and *depositor*; branch-name and balance information is repeated for each holder of an account, but join need not be computed repeatedly.
      - Price paid: more space and more work for programmer to keep relation consistent on updates
    - Better to use materialized views (more on this later..)
  - Cluster together on the same disk page records that would match in a frequently required join
    - Compute join very efficiently when required.

Tuning the Database Design (Cont.)

- **Index tuning**
  - Create appropriate indices to speed up slow queries/updates
  - Speed up slow updates by removing excess indices (tradeoff between queries and updates)
  - Choose type of index (B-tree/hash) appropriate for most frequent types of queries.
  - Choose which index to make clustered
- Index tuning wizards look at past history of queries and updates (the **workload**) and recommend which indices would be best for the workload

Tuning the Database Design (Cont.)

Materialized Views

- Materialized views can help speed up certain queries
  - Particularly aggregate queries
- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update txn
      - time overhead paid by update transaction
    - Deferred view maintenance: done only when required
      - update transaction is not affected, but system time is spent on view maintenance
        - until updated, the view may be out-of-date
- Preferable to denormalized schema since view maintenance is systems responsibility, not programmers
  - Avoids inconsistencies caused by errors in update programs

Tuning the Database Design (Cont.)

- How to choose set of materialized views
  - Helping one transaction type by introducing a materialized view may hurt others
  - Choice of materialized views depends on costs
    - Users often have no idea of actual cost of operations
  - Overall, manual selection of materialized views is tedious
- Some database systems provide tools to help DBA choose views to materialize
  - "Materialized view selection wizards"

Tuning of Transactions

- Basic approaches to tuning of transactions
  - Improve set orientation
  - Reduce lock contention
- Rewriting of queries to improve performance was important in the past, but smart optimizers have made this less important
- Communication overhead and query handling overheads significant part of cost of each call
  - **Combine multiple embedded SQL/ODBC/JDBC queries into a single set-oriented query**
    - Set orientation -> fewer calls to database
    - E.g., tune program that computes total salary for each department using a separate SQL query by instead using a single query that computes total salaries for all department at once (using **group by)**
  - **Use stored procedures**: avoids re-parsing and re-optimization of query

Tuning of Transactions (Cont.)

- Reducing lock contention
- Long transactions (typically read-only) that examine large parts of a relation result in lock contention with update transactions
  - E.g., large query to compute bank statistics and regular bank transactions
- To reduce contention
  - Use multi-version concurrency control
    - E.g., Oracle “snapshots” which support multi-version 2PL
  - Use degree-two consistency (cursor-stability) for long transactions
    - Drawback: result may be approximate

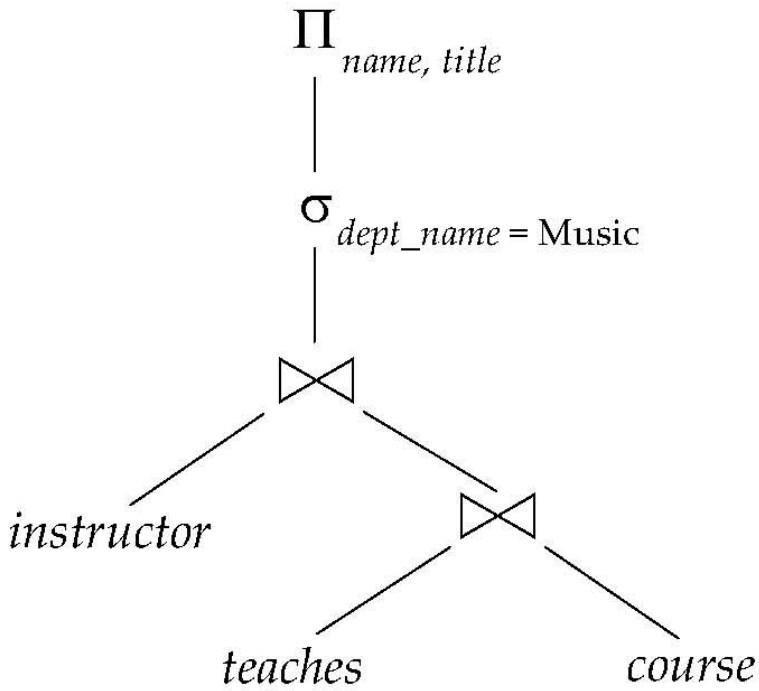
Tuning of Transactions (Cont.)

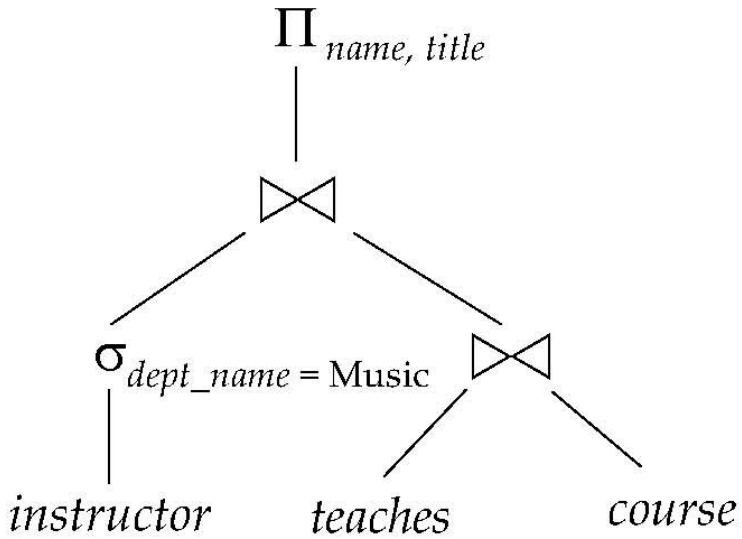
- Long update transactions cause several problems
  - Exhaust lock space
  - Exhaust log space
    - and also greatly increase recovery time after a crash, and may even exhaust log space during recovery if recovery algorithm is badly designed!
- Use **mini-batch** transactions to limit number of updates that a single transaction can carry out. E.g., if a single large transaction updates every record of a very large relation, log may grow too big.
  - Split large transaction into batch of “mini-transactions," each performing part of the updates
  - Hold locks across transactions in a mini-batch to ensure serializability
    - If lock table size is a problem can release locks, but at the cost of serializability
  - In case of failure during a mini-batch, must complete its remaining portion on recovery, to ensure atomicity.

LU46 - Query Optimization: Transformation of Relational Expressions

Introduction

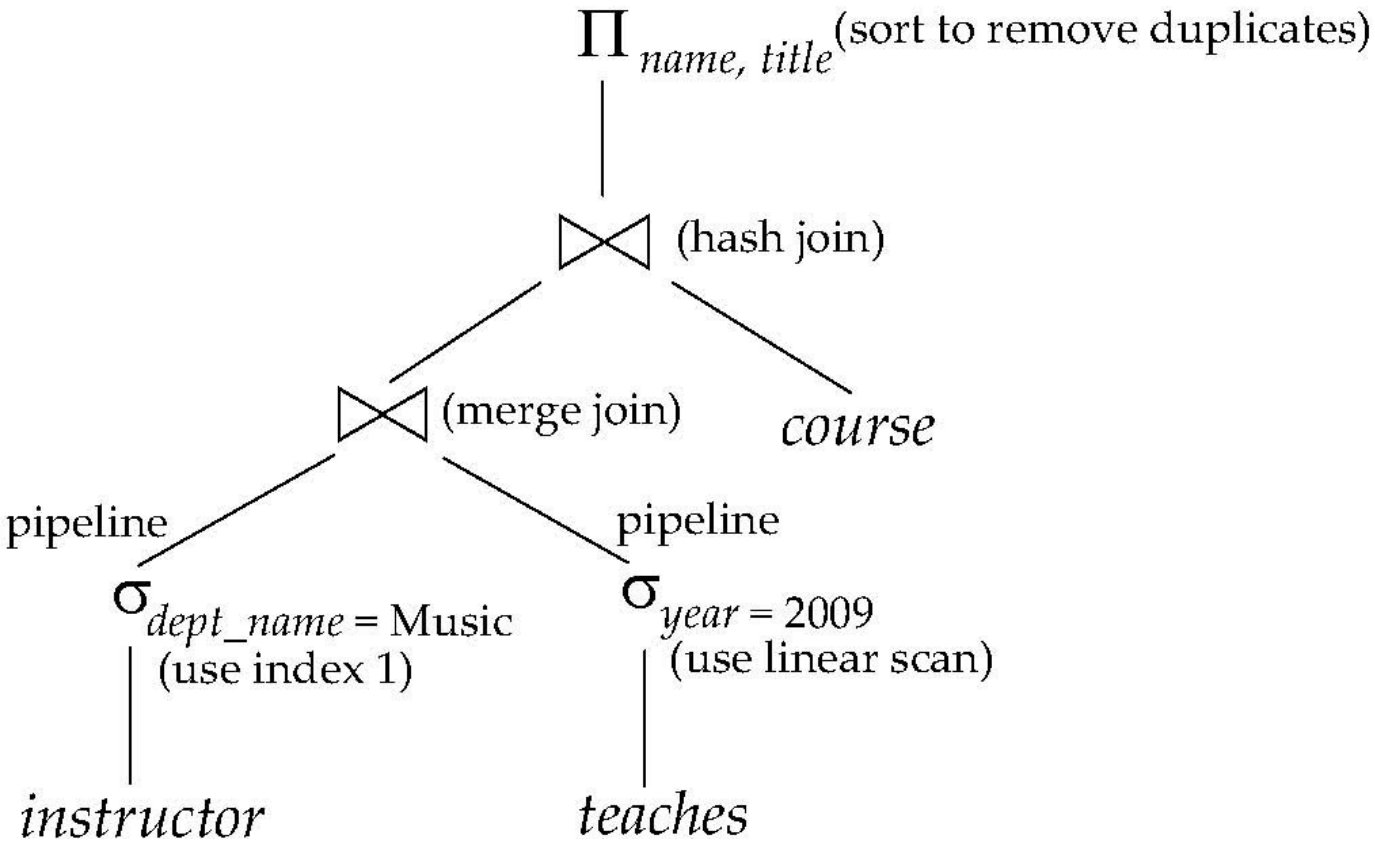
- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation





Introduction (Cont.)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



- Find out how to view query execution plans on your favorite database

Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Viewing Query Evaluation Plans

- Most database support **explain** <query>
  - Displays plan chosen by query optimizer, along with cost estimates
  - Some syntax variations between databases
    - Oracle: **explain plan for** <query> followed by **select** * **from** table (*dbms_xplan.display*)
    - SQL Server: **set showplan_text on**
- Some databases (e.g. PostgreSQL) support **explain analyse** <query>
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as *f..l*
  - *f* is the cost of delivering first tuple and *l* is cost of delivering all results

Generating Equivalent Expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
  - we don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \text{ X } E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

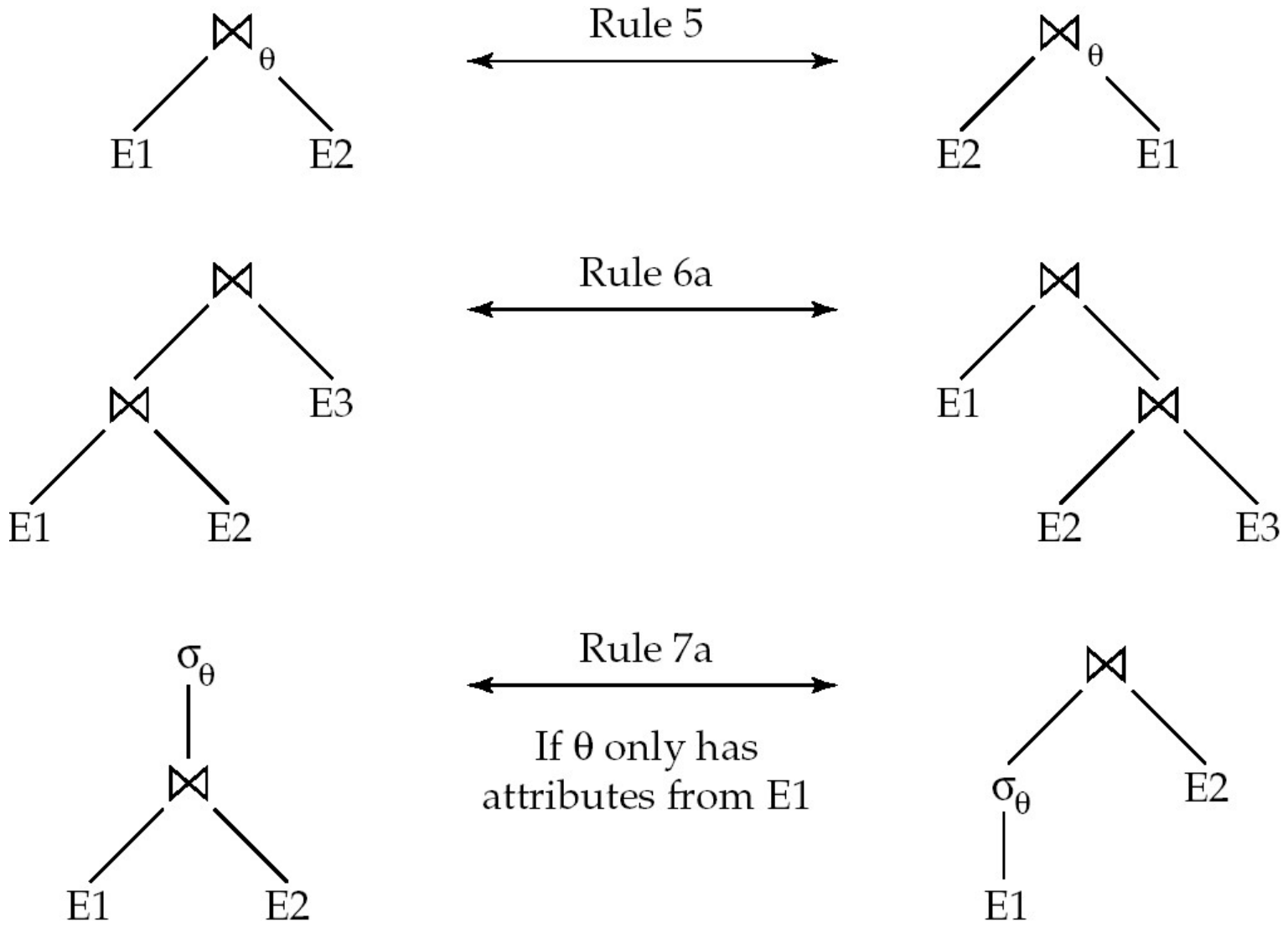
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) 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 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $$\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Exercise

- Create equivalence rules involving
  - The group by/aggregation operation
  - Left outer join operation

Transformation Example: Pushing Selections

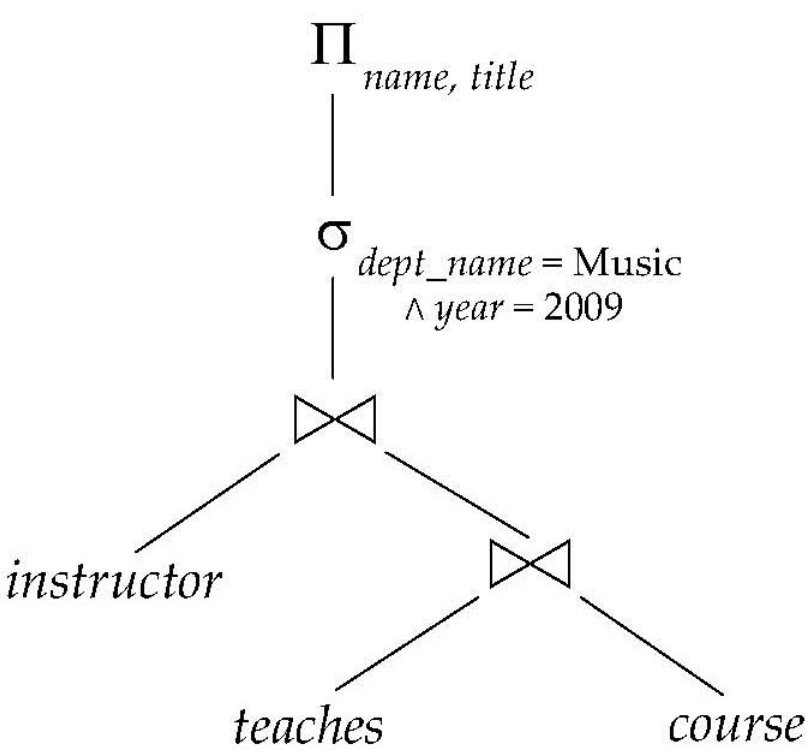
- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{“Music”}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

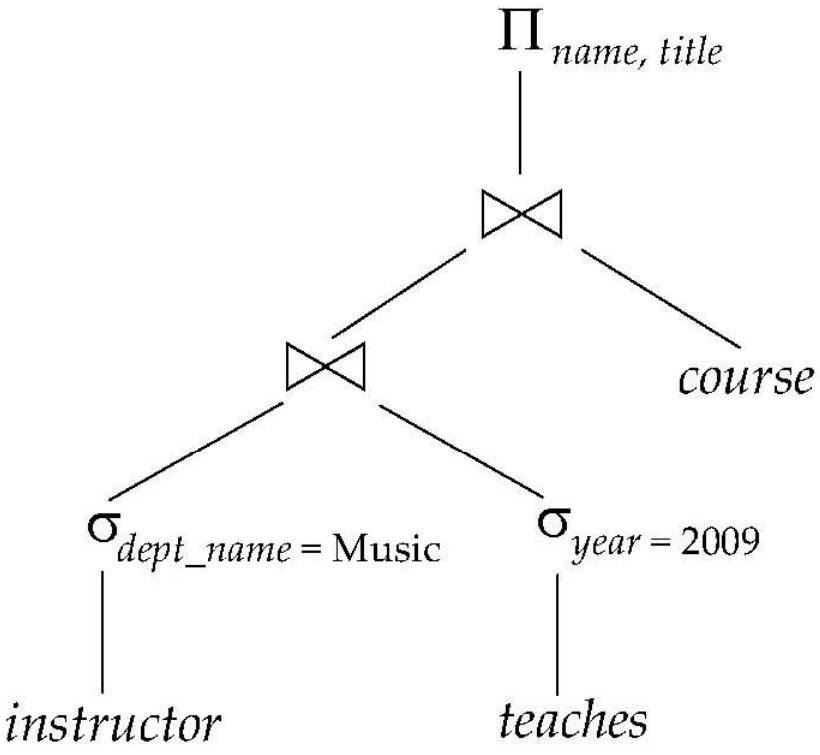
- Query: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge year = 2009}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using join associatively (Rule 6a):
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge gear = 2009}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)))$
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{dept\_name = \text{“Music”}}(instructor) \bowtie \sigma_{year = 2009}(teaches)$$

Multiple Transformations (Cont.)



(a) Initial expression tree



(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$
- When we compute

  $(\sigma_{dept\_name = \text{“Music”}}(instructor \bowtie teaches)$

  we obtain a relation whose schema is:
  (*ID, name, dept_name, salary, course_id, sec_id, semester, year)*
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

  $\Pi_{name,\ title}(\Pi_{name,\ course\_id}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches)) \bowtie \Pi_{course\_id,\ title}(course))))$
- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

  (Join Associativity)

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and join result with

$$\sigma_{dept\_name=\text{“Music”}}(instructor)$$

but the result of the first join is likely to be a large relation.

- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

$$\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches$$

first.

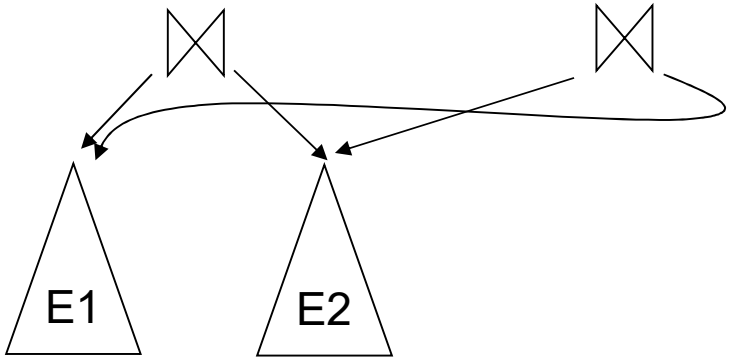
Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers
    - E.g. when applying join commutativity



  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computed as described in Chapter 12
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of *n* relations:
  - To find best plan for a set *S* of *n* relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of *S*.
  - Recursively compute costs for joining subsets of *S* to find the cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
            return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
       set bestplan[S].plan and bestplan[S].cost based on the best way
       of accessing S  using selections on S and indices (if any) on S else for each non-
    empty subset S1 of S such that S1 ≠ S
            P1= findbestplan(S1)
            P2= findbestplan(S - S1)
            for each algorithm A for joining results of P1 and P2
                … compute plan and cost of using A (see next page) ..
                if cost < bestplan[S].cost
                        bestplan[S].cost = cost
                        bestplan[S].plan = plan;
    return bestplan[S]
```

Join Order Optimization Algorithm (cont.)

for each algorithm A for joining results of *P*1 and *P*2
// For indexed-nested loops join, the outer could be *P1* or *P2*
// Similarly for hash-join, the build relation could be *P1* or *P2*
// We assume the alternatives are considered as separate algorithms

if algorithm *A* is indexed nested loops
Let P_i and P_o denote inner and outer inputs
if P_i has a single relation r_i and r_i has an index on the join attribute

plan = "execute P_o.*plan*; join results of P_o and r_i using *A*",
with any selection conditions on P_i performed as part of the join condition
cost = P_o.*cost* + cost of *A*

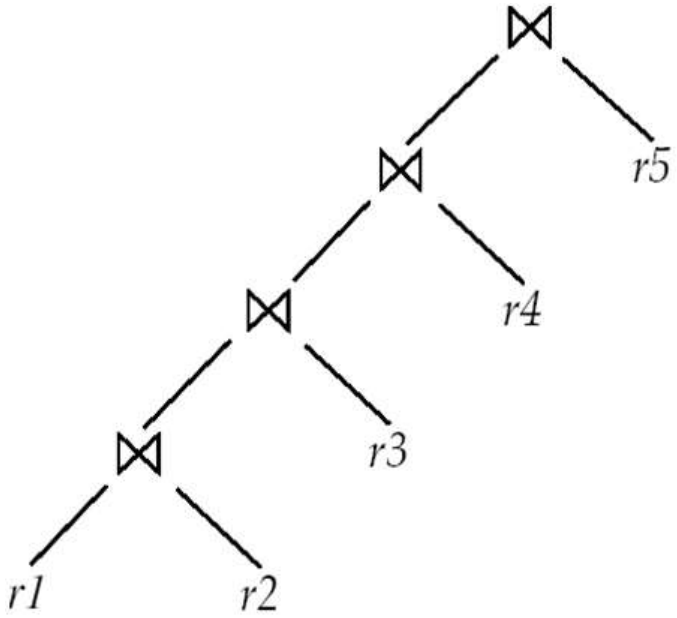
else *cost* = ∞; /* cannot use indexed nested loops join */

else
plan = "execute *P1.plan*; execute *P2.plan*;
join results of *P1* and *P2* using *A*;"
cost = *P1.cost* + *P2.cost* + cost of *A*

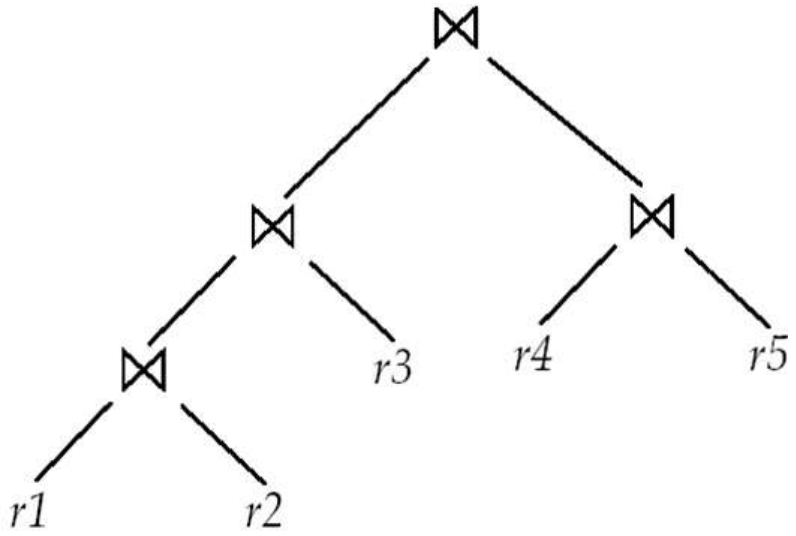
…. See previous page

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree



(b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "**for each** non-empty subset $S1$ of S such that $S1 \neq S$"
    - By: **for each** relation r in S
let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could make a later operation (join/group by/order by) cheaper
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
- Not sufficient to find the best join order for each subset of the set of n given relations
  - must find the best join order for each subset, **for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.
- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on **memoization**, which stores the best plan for a subexpression the first time it is optimized, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans
- Pioneered by the Volcano project and implemented in the SQL Server optimizer

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick “best” relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g., nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
  - **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

LU47 - Estimating Statistics of Expression Results

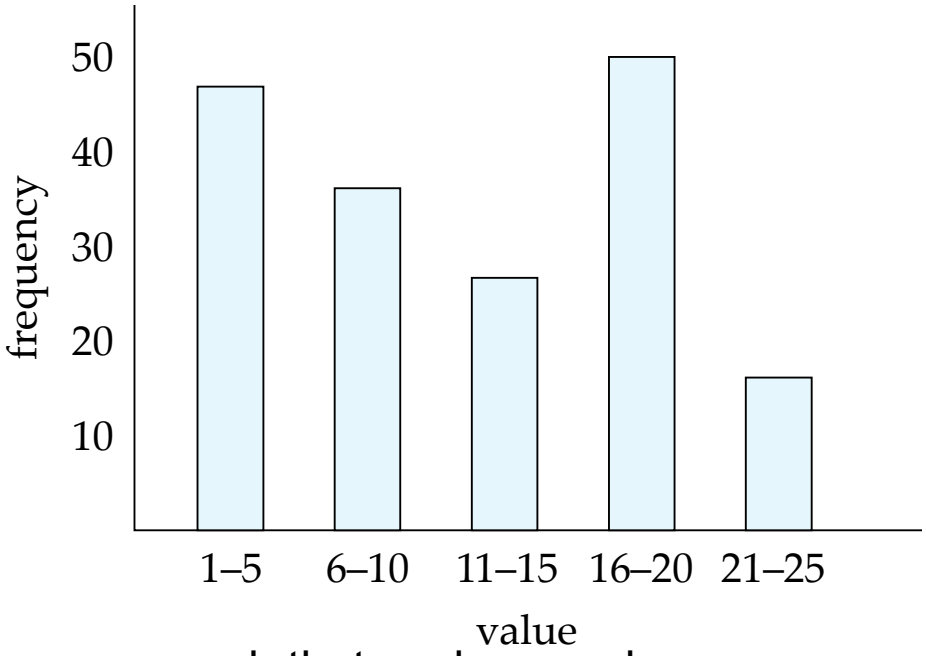
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* **most-frequent values** and their counts
  - Histogram is built on remaining values only

Histograms (cont.)

- Histograms and other statistics usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require a **analyze** command to be executed to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

Selection Size Estimation

- $\sigma_{A=V}(r)$
  - n_r / $V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r \cdot \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .
  - If s_i is the number of satisfying tuples in *r,* the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming independence,* estimate of tuples in the result is: $$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples: $$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$
- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples: $n_r - size(\sigma_\theta(r))$

Join Operation: Running Example

Running example:

$student \bowtie takes$

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*
  - *V*(*ID, student*) = 5000 (*primary key!*)

Estimation of the Size of Joins

- The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *student* $\bowtie$ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(ID, takes)* = 2500, and
    V(ID, student) = 5000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\Pi_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_G\gamma_A(r)$ = $V(G,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g., $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \text{ or } \theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Size Estimation (Cont.)

- Outer join:
  - Estimated size of $r \mathbin{⟕} s$ = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of $r \mathbin{⟕} s$ = *size of* $r \bowtie s$ + size of r + size of s

Estimation of Number of Distinct Values

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., $A = 3$
- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., $(A = 1 \ V \ A = 3 \ V \ A = 4)$),
- If the selection condition θ is of the form $A \ op \ r$
  estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
  - where s is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

- If all attributes in A are from r
  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If A contains attributes $A1$ from r and $A2$ from s, then estimated $V(A, r \bowtie s) =$

  $$\min(V(A1,r)*V(A2 - A1,s),\ V(A1 - A2,r)*V(A2,s),\ n_{r \bowtie s})$$

  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_{A\,(r)}$ as in *r*.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(*A*) and max(*A*), the number of distinct values can be estimated as min(V(*A,r*), *V*(*G,r*)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use *V*(*G,r*)

LU 48 - Choice of Evaluation Plans- Materialized Views

Materialized Views

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  c**reate view** *department_total_salary*(*dept_name, total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*
- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to the materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Incremental maintenance supported by many database systems
    - Avoids manual effort/correctness issues

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i}_r$ and $\mathbf{d}_r$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

A, 1 B, 2	⋈	1, p 2, r 2, s
C,2		

A, 1, p B, 2, r B, 2, s
C, 2, r C, 2, s

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - $R = (A,B)$, and r(R) = { $(a,2)$, $(a,3)$}
  - $\prod_A(r)$ has a single tuple (a).
  - If we delete the tuple $(a,2)$ from r, we should not delete the tuple (a) from $\prod_A(r)$, but if we then delete $(a,3)$ as well, we should delete the tuple
- For each tuple in a projection $\prod_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\prod_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\prod_A(r)$
    - if the count becomes 0, we delete the tuple from $\prod_A(r)$

Aggregation Operations

- **Count** : $v = {}_A\gamma_{count(B)}^{(r)}$.
  - When a set of tuples i_r is inserted
    - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted
    - for each tuple t in i_r we look for the group *t.A* in *v*, and subtract 1 from the count for the group.
      - If the count becomes 0, we delete from *v* the tuple for the group *t.A*
- **Sum**: $v = {}_A\gamma_{sum(B)}^{(r)}$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)

Aggregate Operations (Cont.)

- **Avg**: How to handle average?
  - Maintain **sum** and **count** separately, and divide at the end
- **min**, **max**: $v = {}_{A}\gamma_{min\,(B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of *r* that are in the same group to find the new minimum

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in *r* we check if it is present in *s*, and if so we add it to *v*.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to *s* are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work
  - we leave details to you.

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g., consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that *s* has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace *v* by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: "What is the best set of views to materialize?"
- **Index selection**: "what is the best set of indices to create"
  - closely related, to materialized view selection
    - but simpler
- Materialized view selection and index selection based on typical system **workload** (queries and updates)
  - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
  - One of the steps in database tuning
    - more on tuning in later chapters
- Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

LU49 - Case Studies: PostgreSQL

INTRODUCTION

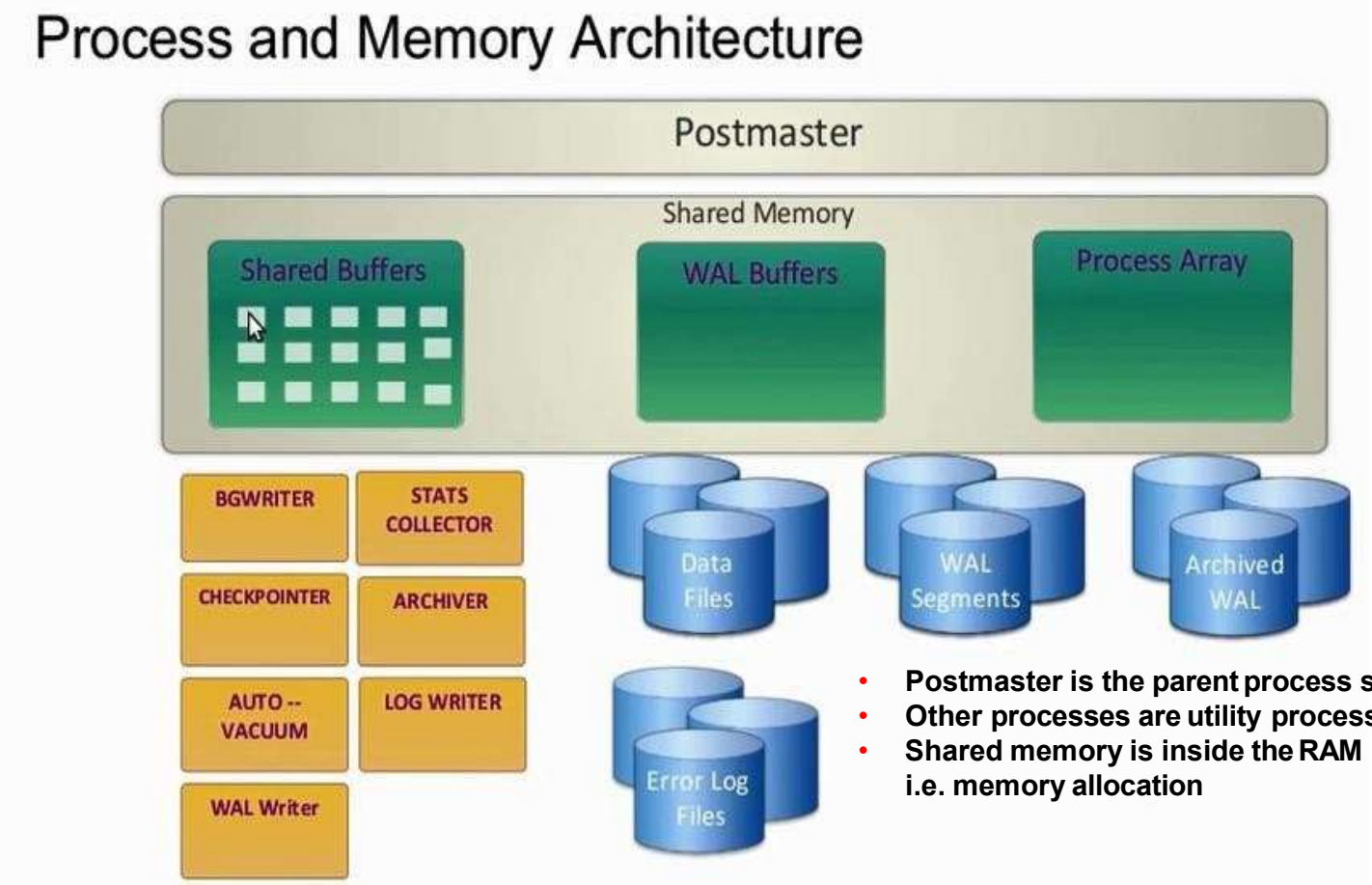
- PostgreSQL, often simply Postgres
- It is an Object-Relational database management system (ORDBMS)
- As a database server, it's primary function is to store data securely, supporting best practices and to allow for data retrieval of other applications.
- Handle workloads from small single-machine applications to large
- internet-facing applications with concurrent users.
- It can handle complex SQL queries using Indexing methods
- Has updateable views and materialized views, triggers, foreign keys.
- Supports functions and stored procedures.
- Cross-platform and runs on many Operating Systems
- Free and Open-Source Software.

HISTORY

- Evolved from the Ingres project at the University of California, Berkeley.
- In 1982, Michael Stone-braker, left Berkeley to make proprietary version of Ingres.
- In 1985 returned to Berkeley and started working on Post-Ingres Project.
- In 1986 POSTGRES team published papers.
- In June 1989 released version 1 to a small number of users.
- Version 2 released in 90, V3 in 91, V4.2 on June 30, 94 with Storage Manager.
- In 95 Postquel replaced by SQL and Front-end program monitor replace by psql.
- The first open-source version was released on Aug 1st, 1996.
- The project was renamed to PostgreSQL and release formed version 6.0 in 97.
- Since then a group of developers and volunteers around the world have maintained the software

  as the PostgreSQL Global Development Group.
- Current version – 13.2

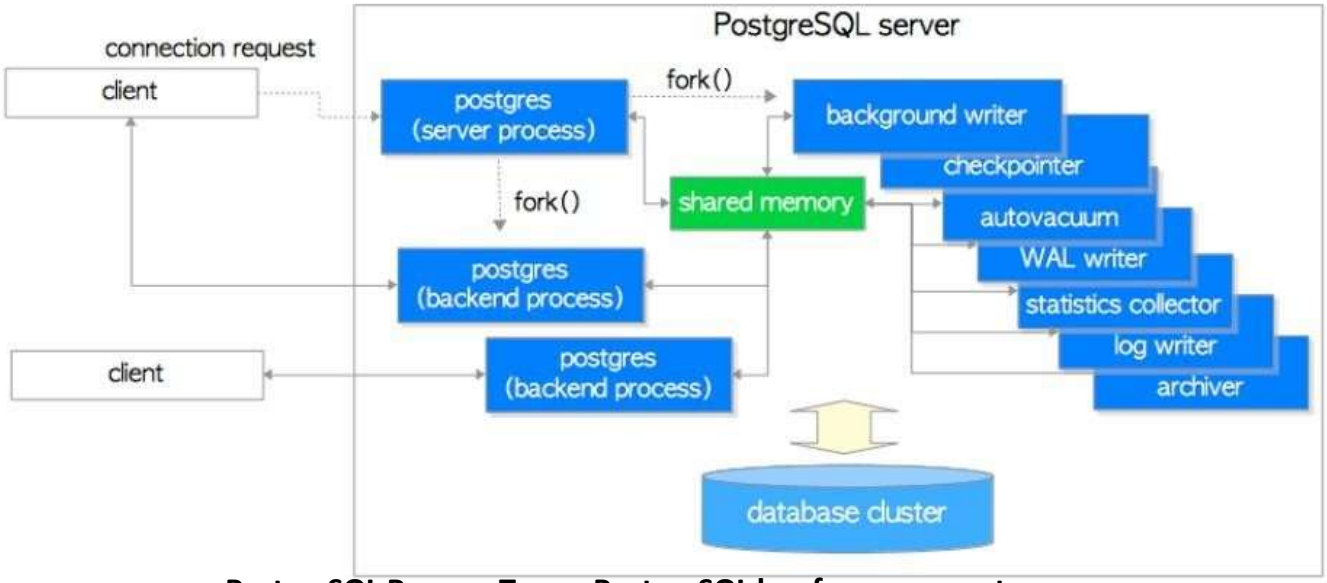
PostgreSQL Architecture



- Postmaster is the parent process supervisor
- Other processes are utility processes
- Shared memory is inside the RAM
  i.e. memory allocation

PostgreSQL Architecture : -

Process architecture in PostgreSQL



PostgreSQL Process Types: PostgreSQL has four process types.

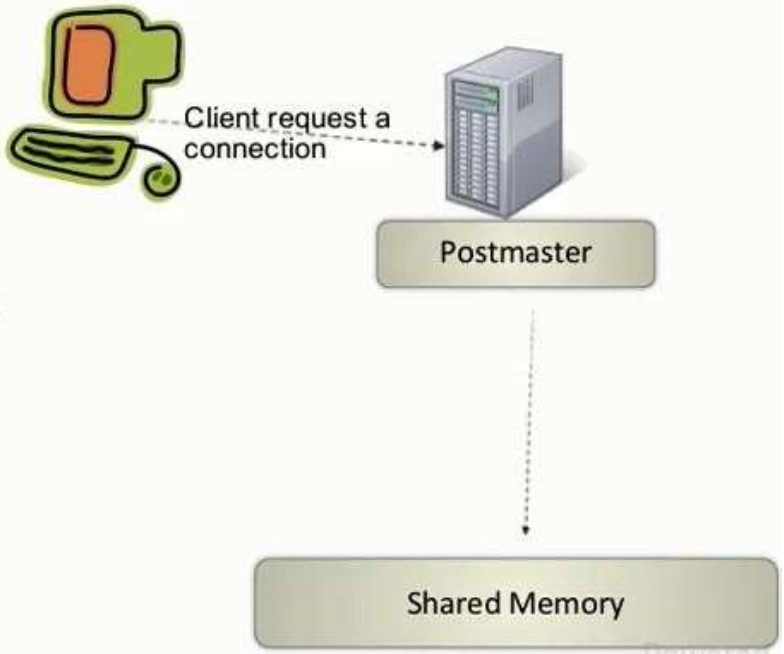
1. **Postmaster (Daemon) Process**
2. **Background Process**
3. **Backend Process**
4. **Client Process**

Background Process

Process	Role
logger	Write the error message to the log file.
checkpointer	When a checkpoint occurs, the dirty buffer is written to the file.
writer	Periodically writes the dirty buffer to a file.
wal writer	Write the WAL buffer to the WAL file.
Autovacuum launcher	Fork autovacuum worker when autovacuum is enabled.It is the responsibility of the autovacuum daemon to carry vacuum operations on bloated tables on demand
archiver	When in Archive.log mode, copy the WAL file to the specified directory.
stats collector	DBMS usage statistics such as session execution information (pg_stat_activity) and table usage statistical information (pg_stat_all_tables) are collected.

Postmaster as Listener

- Postmaster is the master process called postgres
- Listens on 1-and-only-1 tcp port
- Receives client connection request



Query Processing

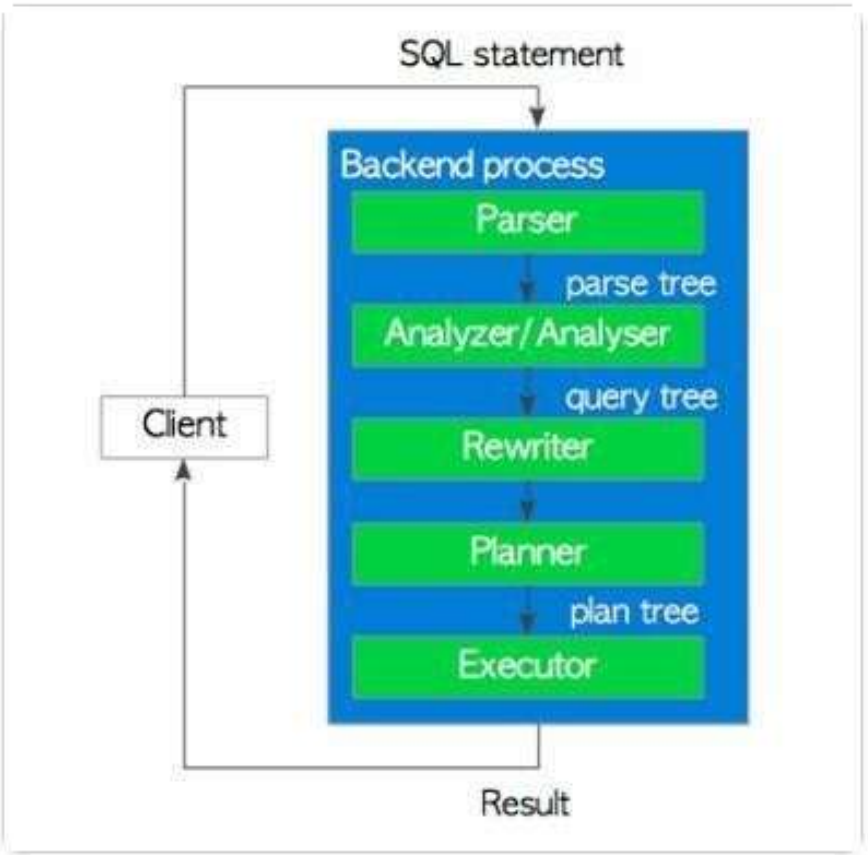
- Parser
- The parser generates a parse tree from an SQL statement in plain text.

- 2. Analyzer/Analyser
- The analyzer/analyser carries out a semantic analysis of a parse tree and generates a query tree.

- 3. Rewriter
- The rewriter transforms a query tree using the rules stored in the rule system if such rules exist.

- 4.Planner
- The planner generates the plan tree that can most effectively be executed from the query tree.

- 5. Executor
- The executor executes the query via accessing the tables and indexes in the order that was created by the plan tree.



Relational Theory

- Data are stored in groups called 'tables' or 'relations'
- Tables consist of 'rows' or 'records' and 'columns' or 'fields'
- Rows are usually identified by a unique 'key' which may be a single column or a group of columns
- Columns can be linked to other tables with similar columns
- Good design of the database structure or 'schema' is critical.
  - "All the data in a row should be dependant on the key, the whole key, and nothing but the key."

Working with databases

- Most of the time, you will work with existing databases
- However, you still need to know the schema
- We will concentrate on four tasks
  - Extracting data
  - Changing data
  - Inserting data
  - Deleting data

PostgreSQL Documentation

- Preface
  - Gives history, conventions, etc.
- Tutorial
  - A quick introduction to SQL
- SQL Language
  - A description of the standard and the PostgreSQL additions
- Reference: SQL Commands
  - Details of each command

Common psql Command Line Options

- -A, --no-align: set output to non-aligned, no padding
- -c *sql*, --command *sql*: execute the sql command and then exit
- -d *name*, --dbname *name*: name of database, same as *name* as the first non-option argument
- -f *name*, --file *name*: use *name* as the source of commands
- -o *name*, --output *name*: put the output in *name*
- -q, --quiet: suppress welcome messages
- -t, --tuples-only: suppress print column names, result row counters, etc
- -?, --help: get command line help

Common psql Meta-commands

- \a: toggle output format (aligned/unaligned)
- \c *name*: connect to a database *name*
- \copy *table*: copy a table to/from a file
- \d : list all tables (display)
- \d *table*: show information about a table
- \e: edit the query buffer
- \g: execute the query buffer (go)
- \h *command*: display help on *command*
- \i *name*: read name into query buffer (input)

Common psql Meta-commands

- \o *name*: send output to *name*
- \p: display the query buffer
- \q: quit the program
- \r: resets (clears) the query buffer
- \t: toggle the output headers on/off
- \w *name*: write the query buffer to *name*
- \! *command*: execute the Linux *command*
- \?: help on meta-commands

Basic Data Types in SQL

- Character, Varchar: A sequence of alphanumeric characters enclosed in single quotes; ‘a string 123’
  - To include a single quote in a string double it; 'Mike''s string'
- Integers: A number without a decimal point; 125, -2
- Real: A number with a decimal point; 3.1416, 25.00
- Decimal, Numeric: Fixed decimal point, accurate but slow math
- Boolean: TRUE or FALSE
  - in psql TRUE is displayed as t, FALSE is displayed as f

Special Values

- TIMESTAMP ‘NOW’
  - The current time, date, or timestamp
- DATE ‘TODAY’
  - The current date or midnight timestamp
- TIMESTAMP ‘TOMORROW’
  - Tomorrow date or midnight timestamp
- DATE ‘YESTERDAY’
  - Yesterday date or midnight timestamp
- TIME ‘ALLBALLS’
  - Midnight 00:00:00 UTC time

Common SQL Commands

- SELECT
  - Extract data from one or more tables
- UPDATE
  - Change data in a table quickly
- INSERT INTO
  - Add new data to a table
- DELETE FROM
  - Remove data from a table
- CREATE TEMP TABLE
  - Create a temporary table
- COPY
  - Send the contents of a table to a file or vice versa

SELECT

- Most SQL work involves creating SELECT statements
- You cannot harm data using a SELECT Command
- SELECT statements can be simple but can also be almost incomprehensible
- SELECT statements are broken up into clauses
  - FROM, WHERE, GROUP BY, HAVING, UNION, INTERSECT, EXCEPT, ORDER BY, LIMIT, FOR UPDATE
- We will only look at a few simple queries

Basic SELECT

- SELECT
  - List of columns to extract
- FROM
  - List of tables to work with
- WHERE
  - Condition to limit selection
- GROUP BY
  - Grouping expression
- HAVING
  - Condition to limit grouping
- ORDER BY
  - Sort order
- LIMIT
  - Number of rows to extract

SELECT Command Structure

SELECT FROM WHERE GROUP BY

List of columns | List of tables | Condition | Expression

HAVING ORDER BY LIMIT

Condition | Sort order | Number

Building a SELECT Query

- You must know the schema (structure) of your database to know the tables and the columns you want to extract
- Know the condition you want to filter the selection
- Know the sort order you want to see the data, the order will be undefined if not specified

Example

Find all IDs and names in Kansas sorted by name

- Tables and columns
  - location: lid, name
  - Table names need to be included only if column names are ambiguous
- Condition
  - location.state = 'KS'
- Order
  - location.name

```
SELECT lid, name FROM location
WHERE state = 'KS'
ORDER BY name;
```

Joining Two Tables

Retrieve the list of locations, ids, and observers in KS sorted by observer's last name

- Tables and columns
  - location: lid, name
  - observer: lastname
- Condition
  - location.lid = observer.lid and location.state = 'KS'
- Sort order
  - observer.lastname

```
SELECT lid, name, lastname FROM location, observer
WHERE location.lid = observer.lid
AND location.state = 'KS'
ORDER BY lastname;
```

Substrings

- Use to extract part of a character column based on position (count from 1)
- SUBSTRING(column FROM begin FOR length)
- Ex. *show phone number of observers in the Kansas City KS Metro area code*
  - SELECT name, hphone FROM observer WHERE SUBSTRING(hphone FROM 1 FOR 3) = '913'

SELECT tricks

- **Testing an expression**
  - SELECT SUBSTRING('ABCDE' FROM 1 FOR 3)
- **Get all the fields in a table**
  - SELECT * FROM location
- **Name shortcut**
  - SELECT l.lid, lastname FROM location l, observer o WHERE l.lid=o.lid AND lastname LIKE 'a%'
- **Rename columns**
  - SELECT lid AS Location_ID FROM location...
- **Eliminate duplicate rows**
  - SELECT DISTINCT county FROM location
- **Order shortcut**
  - SELECT lid, name FROM location ORDER BY 2
- **Descending order**
  - ...ORDER BY lid DESC

Group Functions

- **COUNT()**
  - Instead of displaying the rows, just count them
  - SELECT COUNT(*) FROM observer
- **AVG()**
  - Calculate the average of a column
- **MAX()**
  - Calculate the maximum in a column
- **MIN()**
  - Calculate the minimum in a column
- **SUM()**
  - Calculate the sum of a column

The GROUP BY Clause

- When using the previous functions, the GROUP BY clause tells what to group
- We want the highest crest at EVERY location, sorted by ID
- If we don't use GROUP BY we will get the highest crest of ALL IDs instead of the highest crest at EVERY ID.

```
SELECT l.lid, MAX(stage) FROM location l, crest c
WHERE l.lid = c.lid GROUP BY l.lid ORDER BY l.lid;
```

The UPDATE Command

- Used to quickly change the values in a column
- This command can save a lot of work but can also be dangerous
- Has SET, FROM, and WHERE clauses
- Here is a trick:
  - Create the WHERE clause but use SELECT *
  - You will see the rows you are going to change
  - Build rest of query

The UPDATE Clauses

- SET
  - The column and the value you want to assign it
- FROM
  - A list of tables you used in the SET clause
- WHERE
  - The conditions the data must match

UPDATE Command Structure

UPDATE SET FROM WHERE

Table

Column and value

List of tables

Condition

Using UPDATE

- Table and column
  - observer: hphone
- Value
  - ‘914’ || SUBSTRING(hphone FROM 4)
- Condition
  - SUBSTRING(hphone FOR 3) = '913'

Check WHERE Clause

SELECT * FROM observer WHERE SUBSTRING(hphone FOR 3) = '913'

For every row where hphone starts with '913', the query will replace '913' with '914' while keeping the rest of the string unchanged.

UPDATE observer SET hphone = '914‘ || SUBSTRING(hphone FROM 4)
WHERE SUBSTRING(hphone FOR 3) = '913'

If hphone = '9131234567', after the update, it will become hphone = '9141234567'

The INSERT INTO Command

- Insert data into a new row
- Has only VALUES clause
- Can also use a SELECT query

The INESRT Clause

- VALUES
  - A list of data that you want to insert
  - If the order of the columns is not listed, it must match the order defined in the schema
  - For safety sake, always list the columns
- SELECT
  - A standard select query, but the schema of the result must match the target

INSERT Command Structure

INSERT INTO VALUES

Table, column list

List of values

-or-

INSERT INTO SELECT

Table, column list

Select query

Using INSERT

- Want to add a new row in datum
- datum schema is
  - lid - location id
  - ddate - datum date
  - elev - elevation
- Our values are ABEK1, 2002-3-23, 3056.34

```
INSERT INTO datum (lid, ddate, elev)
VALUES ('ABEK1', '2002-3-23', 3056.34);
```

- Want to add data from a temporary table named junk into the height table
- junk and height have the same schema

```
INSERT INTO height SELECT * FROM junk;
```

The DELETE FROM Command

- Remove rows
- Has only WHERE clause
- There is no quick undo so be careful
- If you accidentally leave off the WHERE clause, ALL the rows will be deleted!
- Here is another trick:
  - Create the WHERE clause but use SELECT *
  - You will see the rows you are going to delete
  - Change SELECT * to DELETE
- WHERE
  - The condition the rows must match

DELETE Command Structure

DELETE FROM WHERE

Table Condition

Using DELETE

- Want to remove an observer to make a location inactive
- The location ID is CANK1

Check WHERE Clause

```
SELECT * FROM observer WHERE lid = 'CANK1'
```

DELETE row

```
DELETE FROM observer WHERE lid = 'CANK1'
```

The COPY Command

- There is a quick and easy way to copy a table to a file
- You could process a table without working in SQL
- This is similar to the Informix UNLOAD command
- It has many options but we will only look at a few
- It combines well with the CREATE TEMP TABLE command
- You must be logged on as *postgres* for this command to work

COPY Command Structure

COPY Table FROM File TO WITH Options

Example COPY Commands

- COPY observer TO 'observer.dat' WITH DELIMITER '|';
  - Same as Informix UNLOAD command except default delimiter is a TAB
- COPY crest TO 'crest.csv' WITH CSV;
  - Good way of moving data into Excel
- COPY observer FROM 'observer.dat' WITH DELIMITER '|';
  - Moving data into a table

TABLE and COPY

- We want to copy today's stages into a file, stg_tdy.dat

```
CREATE TEMP TABLE junk AS
SELECT * FROM height
WHERE AGE(obstime) < INTERVAL '1 day';
COPY junk TO stg_tdy.dat;
```

In Conclusion

- PostgreSQL may not be as popular as MySQL, it will do what we need it to do
- The PostgreSQL documentation is excellent and available on-line as HTML or off-line as PDF
- psql is good to use in scripts
- Snoopy is good to use interactively
- The best way to learn SQL is to play with it
- You cannot harm data by SELECT queries