



Query Processing, optimization, and indexing techniques



What's this tutorial about?

- From here:

```
SELECT C.name AS Course, count(S.students) AS Cnt
FROM courses C, subscription S
WHERE
```

```
    C.lecturer = "Calders"
```

```
    AND C.courseID = S.courseID
```

- To there:

Course	Cnt
"Advanced Databases"	67
"Data mining en kennissystemen"	19

- What's in between?
- How does a *relational* DBMS get there *efficiently*.



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Physical Reality

- Cost of query evaluation 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*
 - ▶ Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



What's this tutorial about?

- Factors that influence the efficiency:
 - How is the data stored?
 - ▶ Primary and secondary indices
 - ▶ B-trees
 - ▶ Composite search keys
 - ▶ Hashing
 - How is the query processed?
 - ▶ Relational algebra
 - ▶ Query evaluation plan
- We start with the second part ...

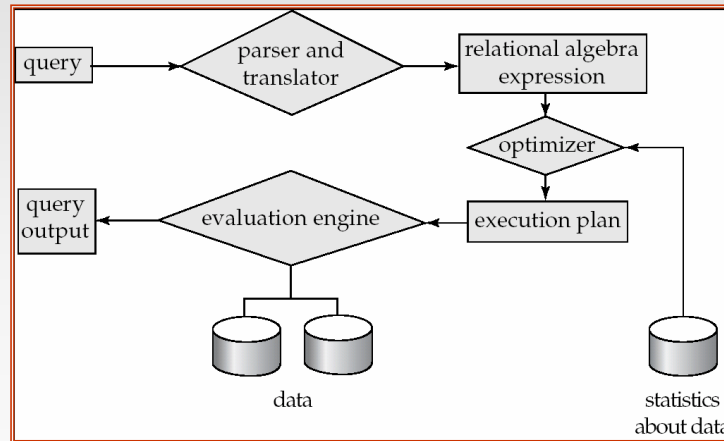
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

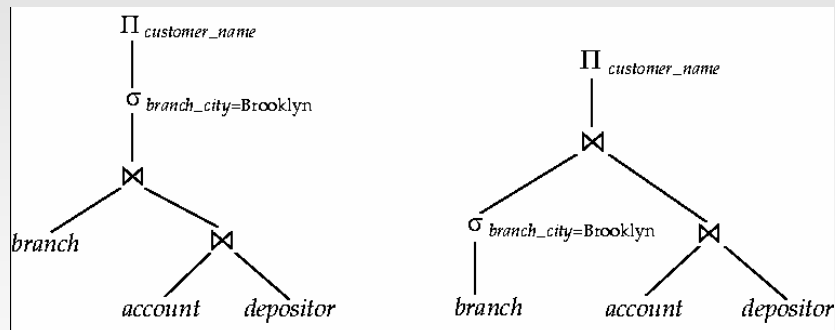


Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Logical Query Plan

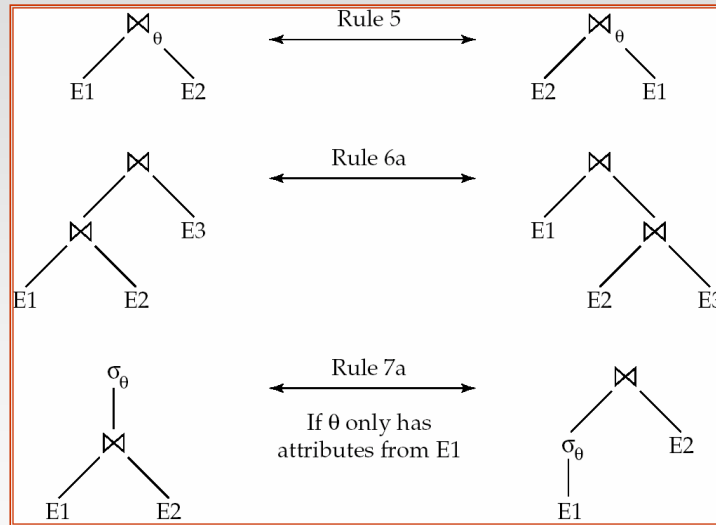
- SQL query is translated into a relational algebra expression
 - can be seen as a tree
 - different expressions are possible



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



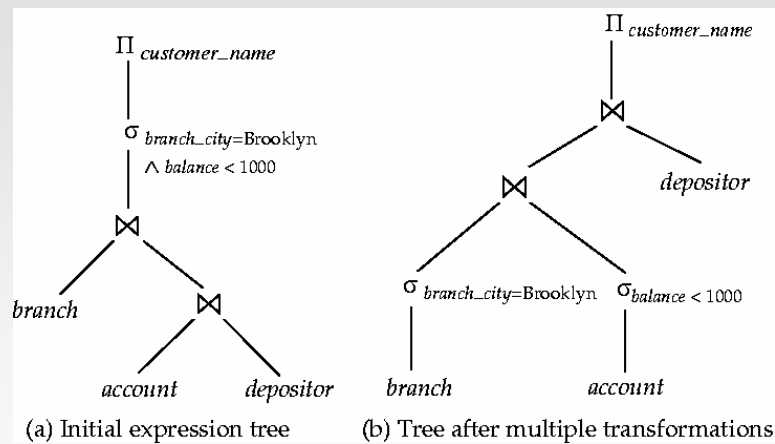
Pictorial Depiction of Equivalence Rules



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Multiple Transformations (Cont.)



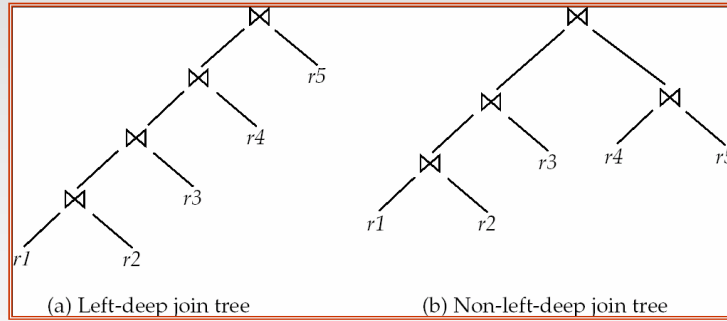
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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.



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Physical Query Plan

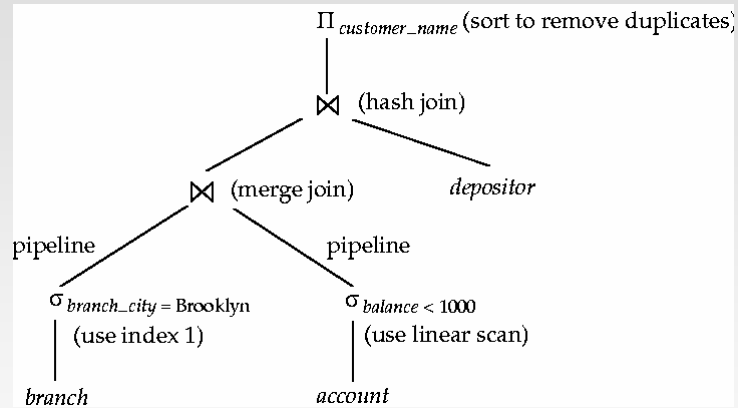
- For all relational algebra expressions
 - Different implementations
- Best choice highly depends on
 - Number of tuples
 - Presence or absence of indices (way of storage)
 - Selectivity of predicates (statistics!)
 - Pipelined or materialized
 - Etc...
- Physical query plan = logical query plan + choice of implementation

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Physical Query Plan



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Optimization

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
- Cost is estimated
 - using statistical information from the database catalog
 - ▶ number of tuples in each relation,
 - ▶ size of tuples,
 - based on the way the data is stored
 - ▶ ordered w.r.t. the primary key or not
 - ▶ secondary indices

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Indexing Structures



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Multiple-Key Access



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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”.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.

Brighton		→	A-217	Brighton	750	
Downtown		→	A-101	Downtown	500	
Mianus		→	A-110	Downtown	600	
Perryridge		→	A-215	Mianus	700	
Redwood		→	A-102	Perryridge	400	
Round Hill		→	A-201	Perryridge	900	
			A-218	Perryridge	700	
			A-222	Redwood	700	
			A-305	Round Hill	350	

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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

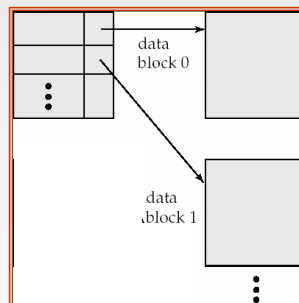
Brighton		A-217	Brighton	750	
Mianus		A-101	Downtown	500	
Redwood		A-110	Downtown	600	
		A-215	Mianus	700	
		A-102	Perryridge	400	
		A-201	Perryridge	900	
		A-218	Perryridge	700	
		A-222	Redwood	700	
		A-305	Round Hill	350	

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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.



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





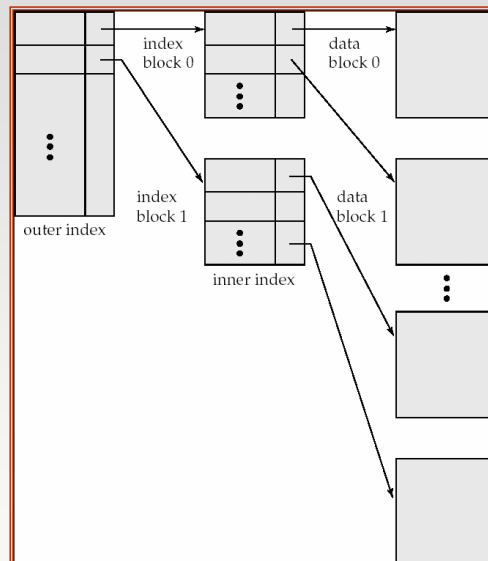
Multilevel Index

- 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.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Multilevel Index (Cont.)



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





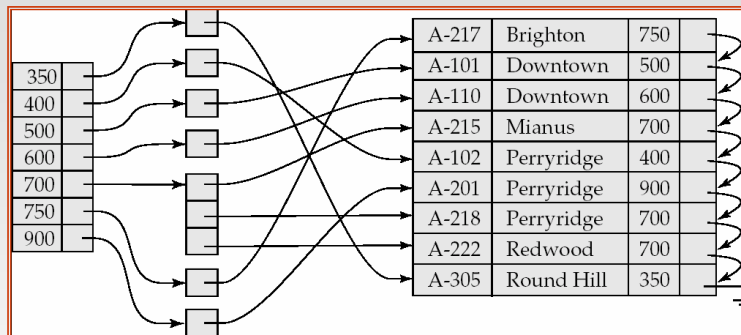
Secondary Indices

- 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 *account* relation stored sequentially by account number, we may want to find all accounts in a particular branch
 - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Secondary Indices Example



Secondary index on *balance* field of *account*

- 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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



B+-Tree Node Structure

- Typical node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- 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

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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.

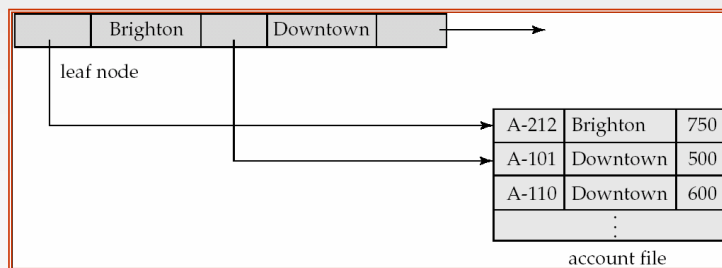




Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i either points to a file record with search-key value K_i , or to a bucket of pointers to file records, each record having search-key value K_i . Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than L_j 's search-key values
- P_n points to next leaf node in search-key order



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Non-Leaf Nodes in B+-Trees

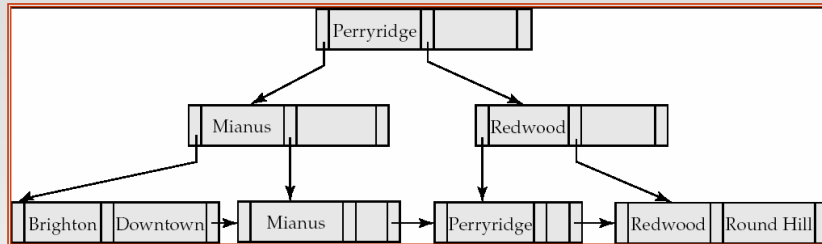
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the search-keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to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



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Example of a B⁺-tree

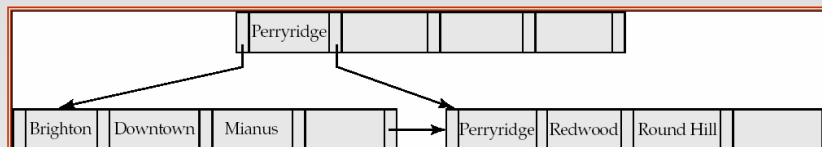


B⁺-tree for *account* file ($n = 3$)

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Example of B⁺-tree



B⁺-tree for *account* file ($n = 5$)

- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).
- Root must have at least 2 children.

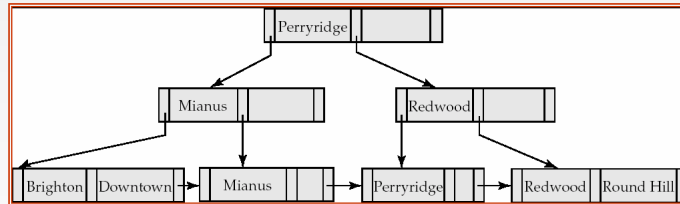
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Queries on B+-Trees

- Find all records with a search-key value of k .
 1. $N = \text{root}$
 2. Repeat
 1. Examine N for the smallest search-key value $> k$.
 2. If such a value exists, assume it is K_i . Then set $N = P_i$
 3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$
 Until N is a leaf node
 3. If for some i , key $K_i = k$ follow pointer P_i to the desired record or bucket.
 4. Else no record with search-key value k exists.



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Queries on B+-Trees (Cont.)

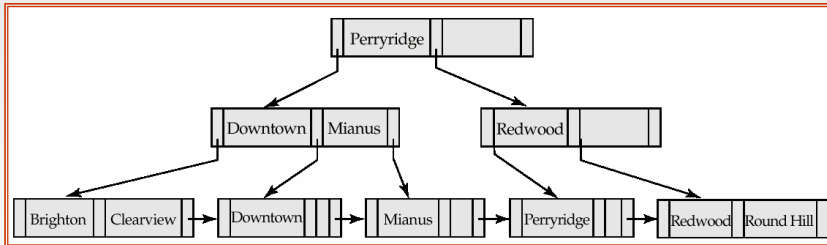
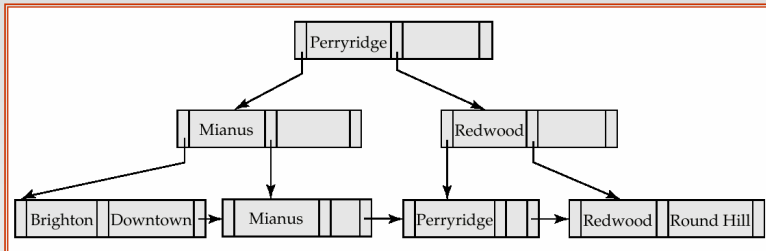
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
 - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values 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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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

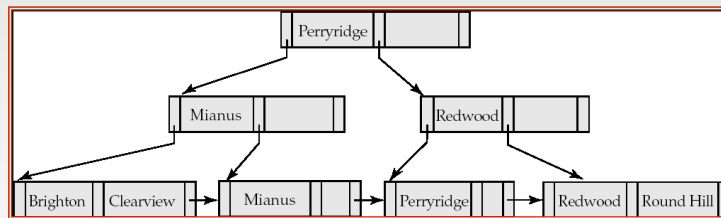
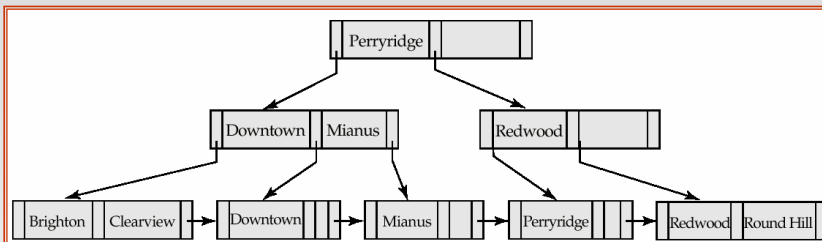


B+-Tree before and after insertion of "Clearview"

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Examples of B+-Tree Deletion



Before and after deleting "Downtown"

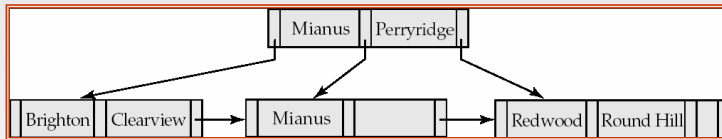
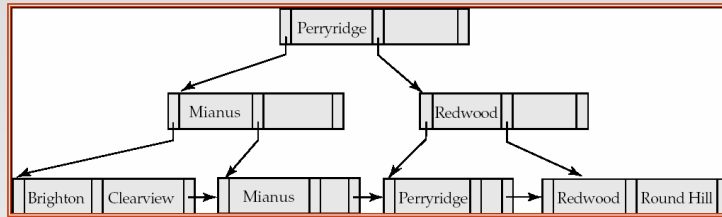
- Deleting "Downtown" causes merging of under-full leaves
 - leaf node can become empty only for $n=3!$

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Examples of B+-Tree Deletion (Cont.)



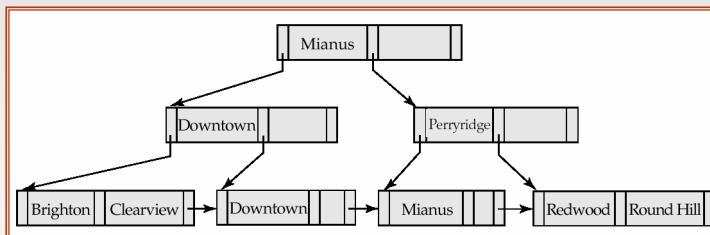
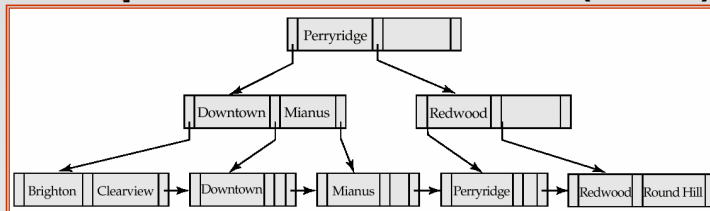
Deletion of "Perryridge" from result of previous example

- Leaf with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling
 - Value separating two nodes (at parent) moves into merged node
 - Entry deleted from parent
- Root node then has only one child, and is deleted

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Example of B+-tree Deletion (Cont.)



Before and after deletion of "Perryridge" from earlier example

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





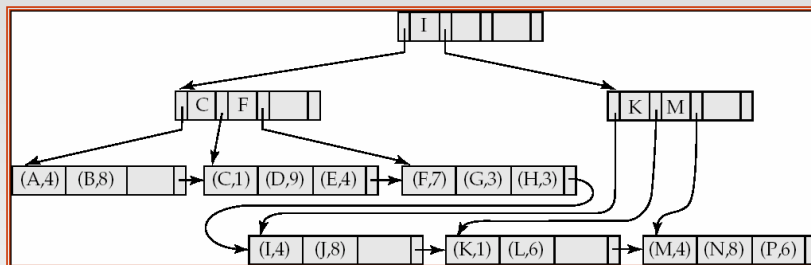
B+-Tree File Organization

- 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.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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

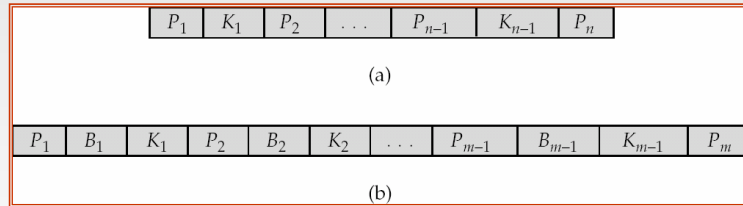
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

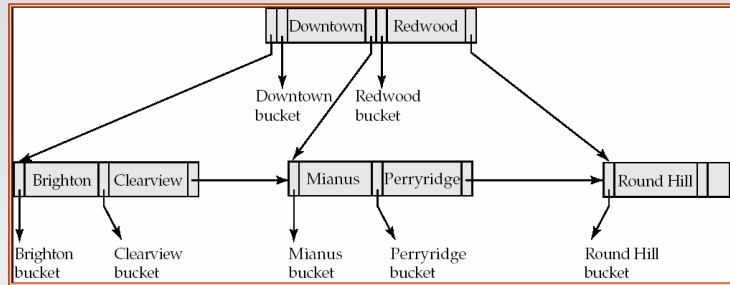


- Nonleaf node – pointers B_i are the bucket or file record pointers.

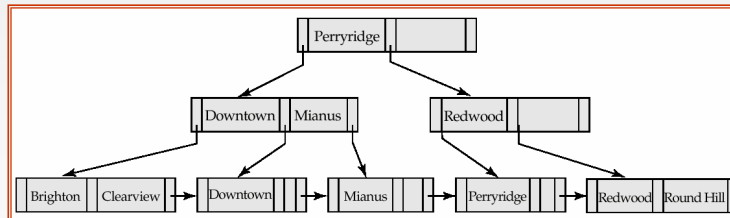
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

```
select account_number  
from account  
where branch_name = "Perryridge" and balance = 1000
```
- Possible strategies for processing query using indices on single attributes:
 1. Use index on *branch_name* to find accounts with branch name Perryridge; test *balance = 1000*
 2. Use index on *balance* to find accounts with balances of \$1000; test *branch_name = "Perryridge"*.
 3. Use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
 - E.g. (*branch_name*, *balance*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
 - $a_1 < b_1$, or
 - $a_1 = b_1$ and $a_2 < b_2$

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Implementations of Relational Algebra Expressions



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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.
 - **A2** (*binary search*). Applicable if selection is an equality comparison on the attribute on which file is ordered.
 - **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
 - **A4** (*primary index on nonkey, equality*) Retrieve multiple records.
 - **A5** (*equality on search-key of secondary index*).
 - **A6** (*primary index, comparison*). (Relation is sorted on A)
 - **A7** (*secondary index, comparison*)
 - ...



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



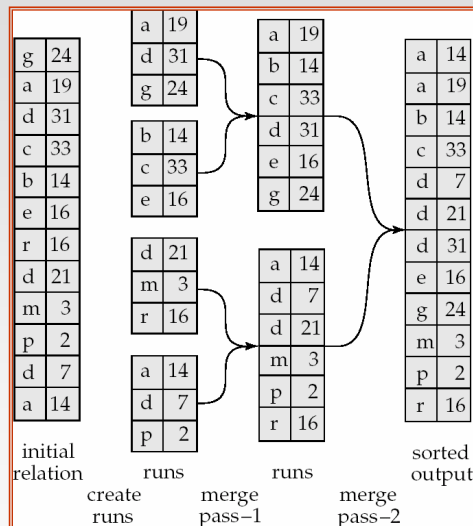
Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Example: External Sorting Using Sort-Merge



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Join Operation

- Several different algorithms to implement joins
 - Nested-loop join
 - Block nested-loop join
 - Indexed nested-loop join
 - Merge-join
 - Hash-join
- Choice based on cost estimate
- Examples use the following information
 - Number of records of *customer*: 10,000 *depositor*: 5000
 - Number of blocks of *customer*: 400 *depositor*: 100

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Nested-Loop Join

- 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 $t_r \cdot t_s$ to the result.
 - end**
 - end**
- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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  $t_r \cdot t_s$  to the result.
      end
    end
  end
end
```

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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
 - ▶ Can construct an index just to compute a join.
- For each tuple t_r in the outer relation r , use the index 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 .

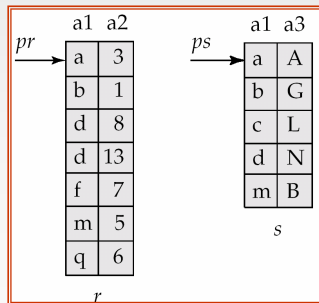
Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





Merge-Join

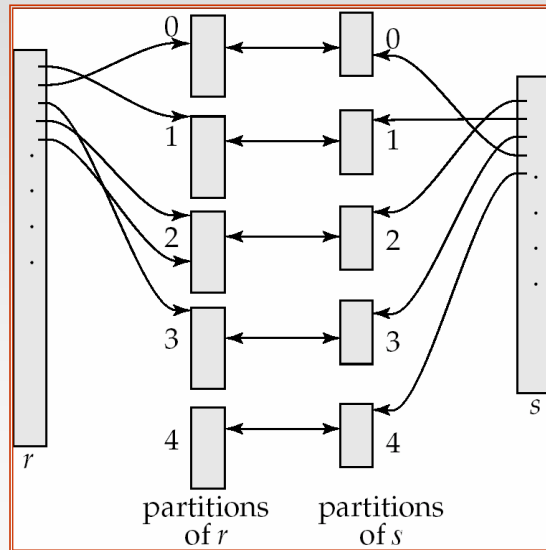
1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
 1. Join step is similar to the merge stage of the sort-merge algorithm.
 2. Main difference is handling of duplicate values in join attribute — every pair with same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



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Hash-Join (Cont.)



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





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**.

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



Evaluation of Expressions

- So far: we have seen algorithms for individual 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

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.





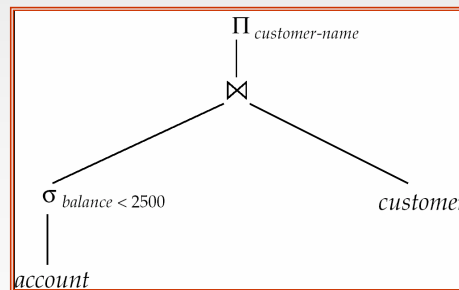
Materialization

- **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

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

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



Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.



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.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- 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**

Based upon slides for: Database System Concepts - 5th Edition, Aug 27, 2005.

