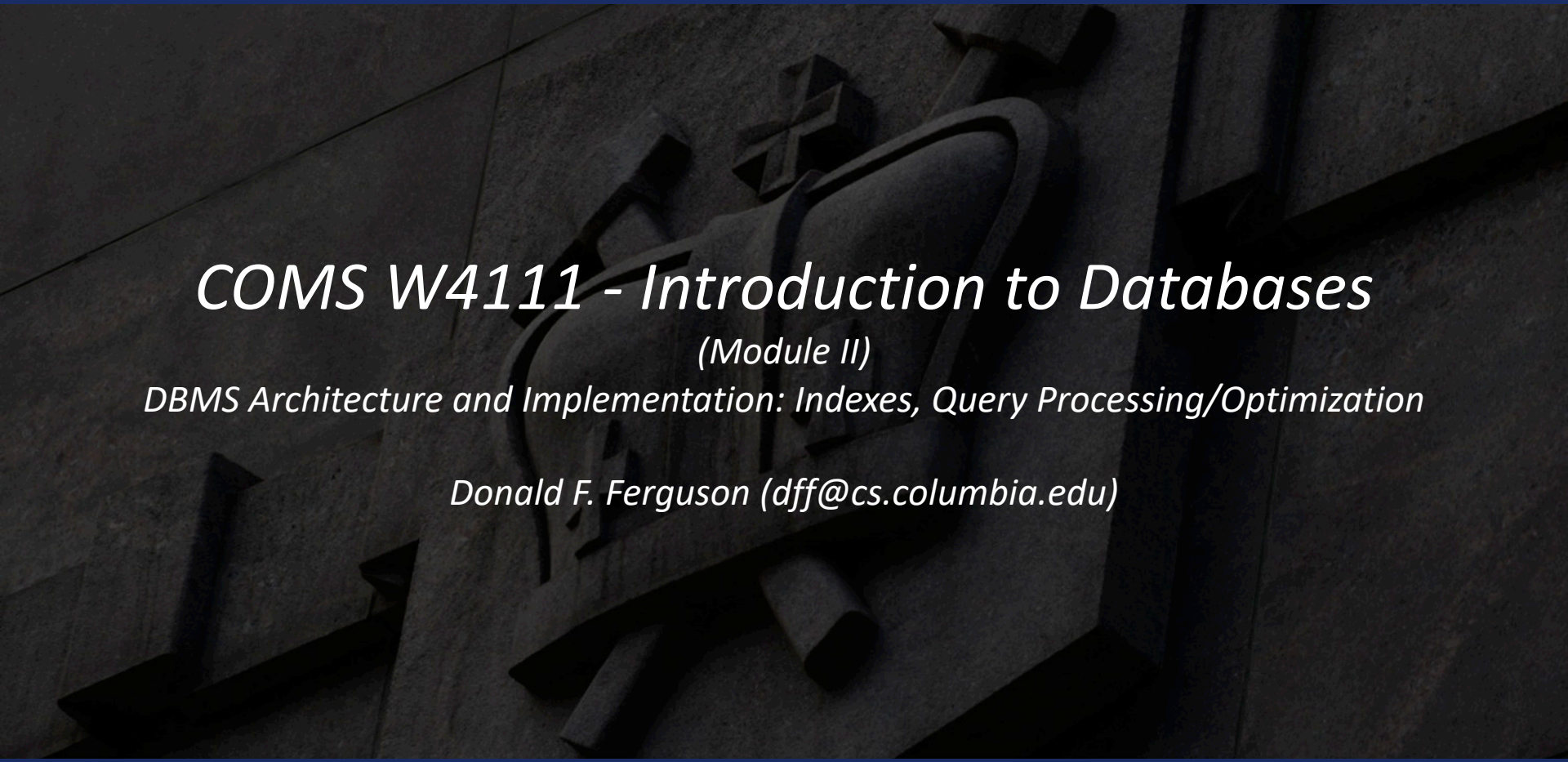


# COMS W4111 - Introduction to Databases

*(Module II)*

*DBMS Architecture and Implementation: Indexes, Query Processing/Optimization*

*Donald F. Ferguson (dff@cs.columbia.edu)*

# *Reminder*

# Database Management System Reminder

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

**Database Systems: The Complete Book (2nd Edition)**
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

# Database Management System Reminder

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

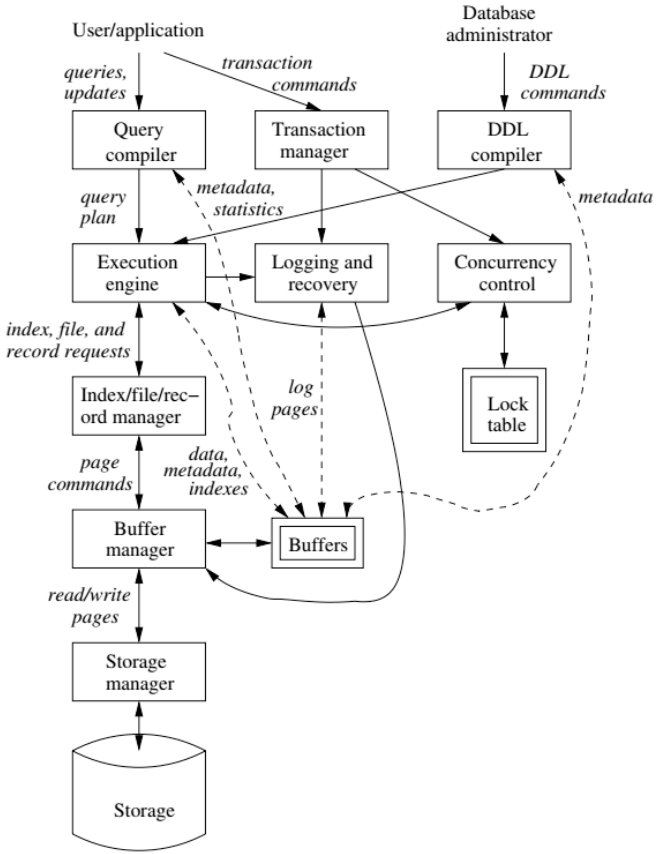
4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

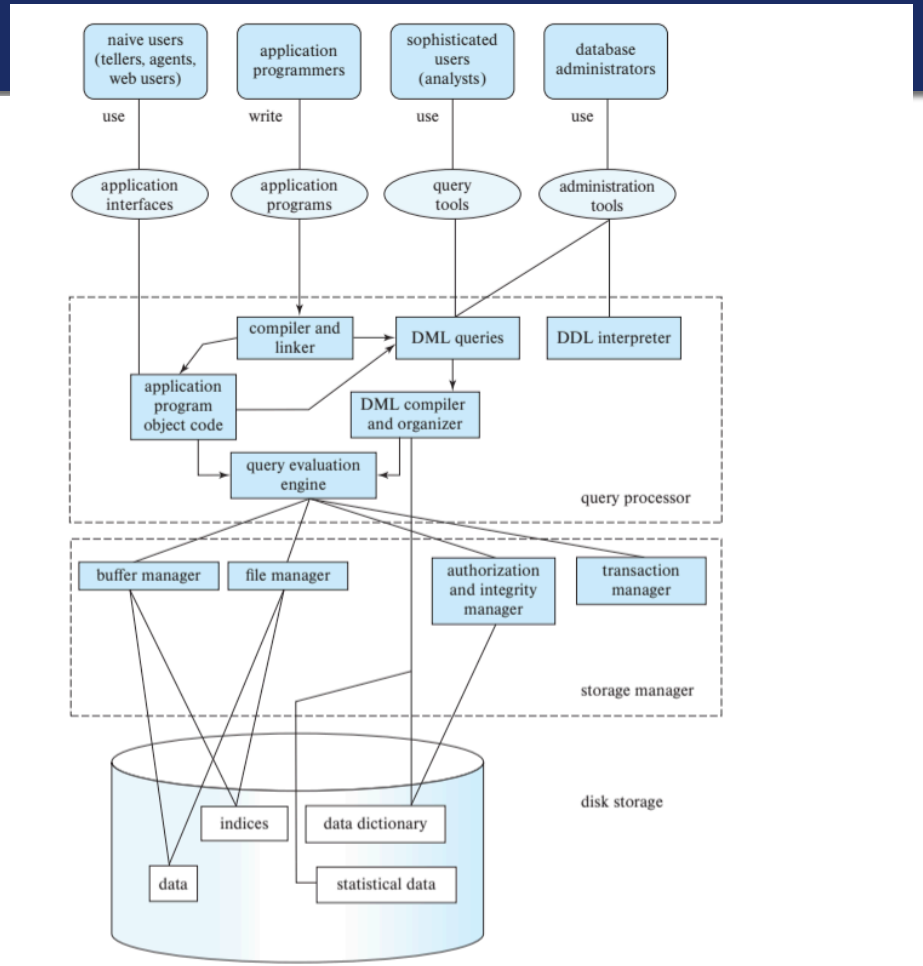
5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Focus for next part of course.

**Database Systems: The Complete Book (2nd Edition)**
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

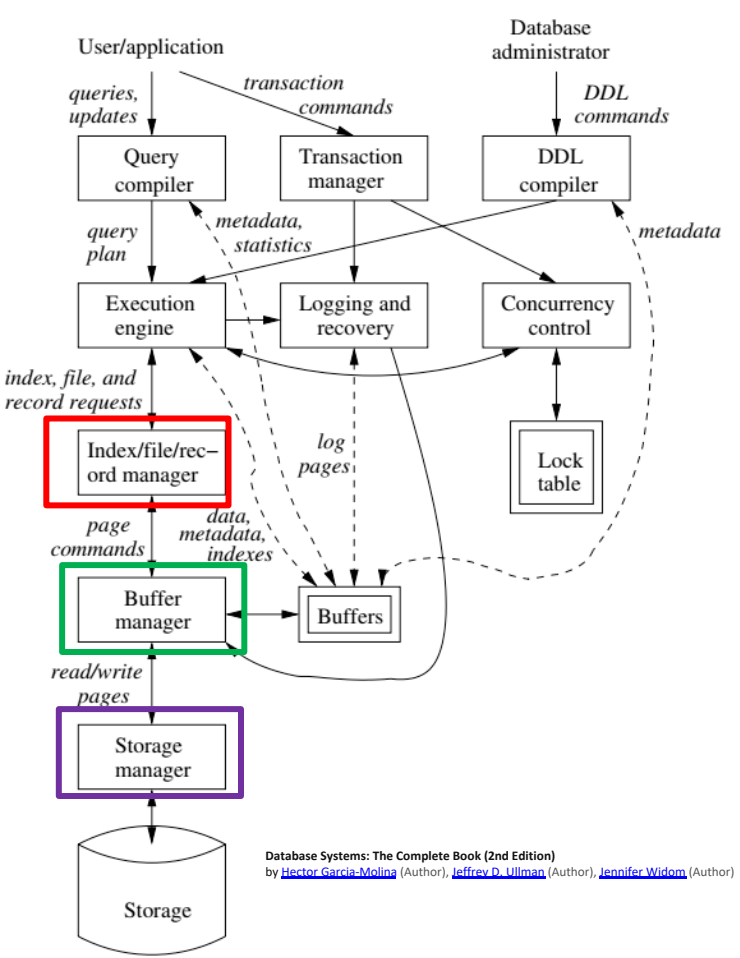
# DBMS Subsystems



# Data Management

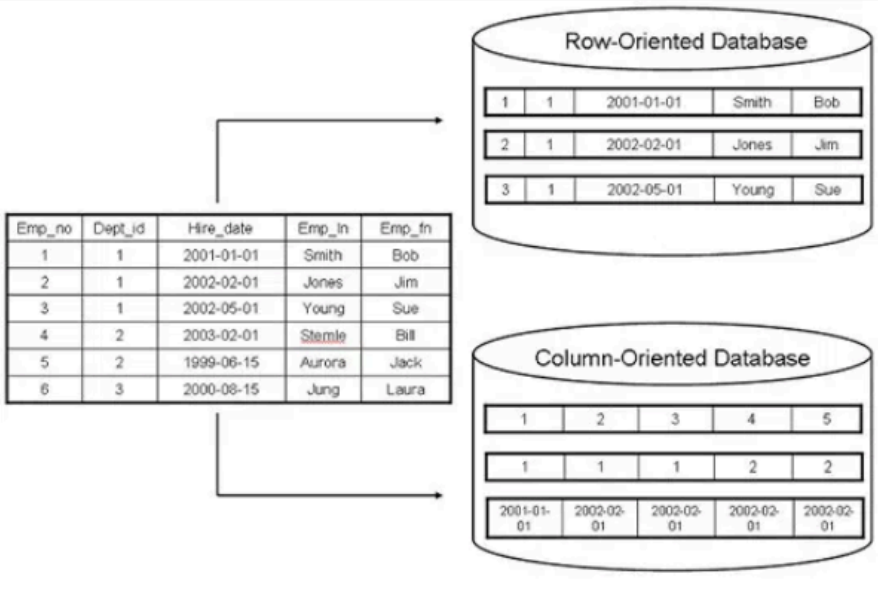
- Find things quickly.
- Access things quickly.
- Load/save things quickly.




Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

# *Some Final Thoughts on Disks and Buffers*

# Row versus Columnar Storage




Column-oriented data storage allows us to access all of the entries in a database column quickly and efficiently. Columnar storage formats are mostly relevant today for performing large analytics jobs.

https://softwareengineeringdaily.com/2017/01/13/columnar-data-apache-arrow-and-parquet-with-julien-le-dem-and-jacques-nadeau/

# Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

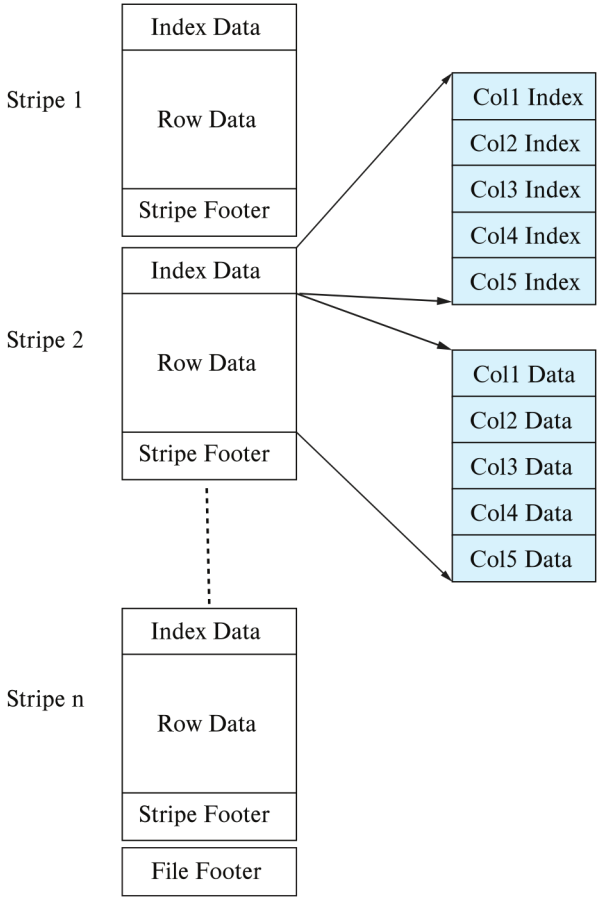
| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 60000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri | History | 62000 |
| 76543 | Singh | Finance | 80000 |
| 76766 | Crick | Biology | 72000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |

# Columnar Representation

- Benefits:
  - Reduced IO if only some attributes are accessed
  - Improved CPU cache performance
  - Improved compression
  - **Vector processing** on modern CPU architectures
- Drawbacks
  - Cost of tuple reconstruction from columnar representation
  - Cost of tuple deletion and update
  - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
  - Called **hybrid row/column stores**

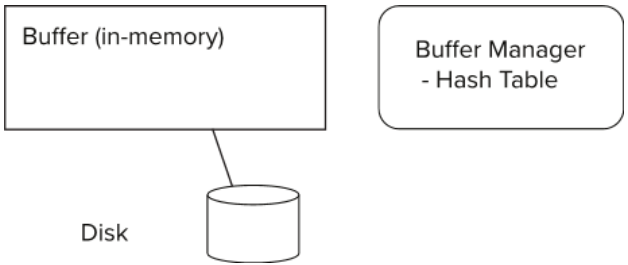
# Columnar File Representation

- ORC and Parquet: file formats with columnar storage inside file
- Very popular for big-data applications
- Orc file format shown on right:



# Storage Access

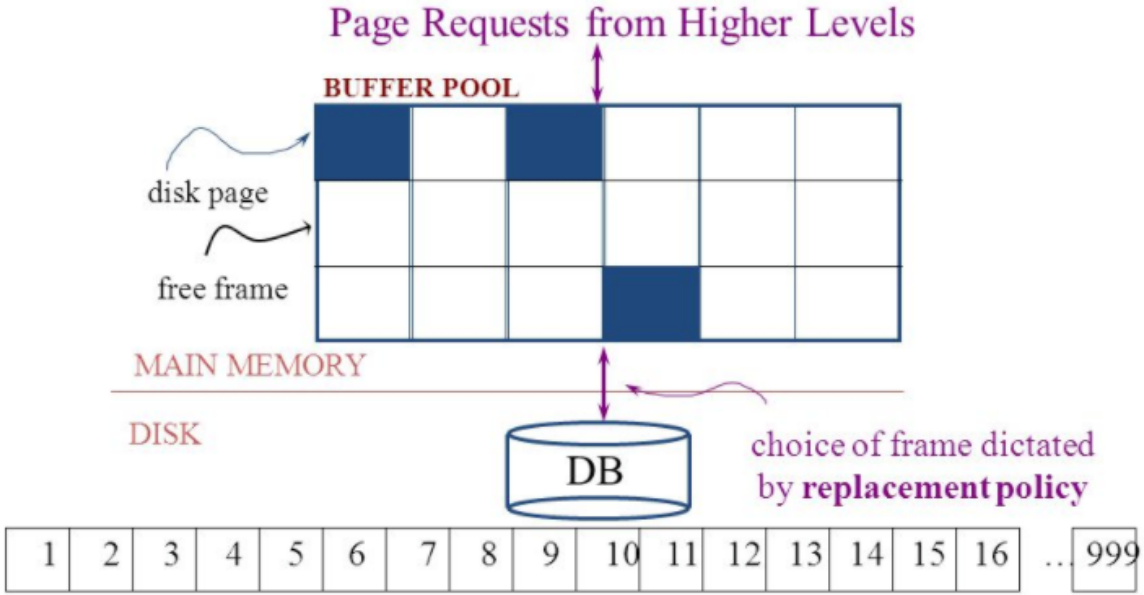
- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



# The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.



# Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

# Buffer Manager

- **Buffer replacement strategy** (details coming up!)
- **Pinned block:** memory block that is not allowed to be written back to disk
  - **Pin** done before reading/writing data from a block
  - **Unpin** done when read /write is complete
  - Multiple concurrent pin/unpin operations possible
    - Keep a pin count, buffer block can be evicted only if pin count = 0
- **Shared and exclusive locks on buffer**
  - Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
  - Readers get shared lock, updates to a block require exclusive lock
  - **Locking rules:**
    - Only one process can get exclusive lock at a time
    - Shared lock cannot be concurrently with exclusive lock
    - Multiple processes may be given shared lock concurrently

# Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - Idea behind LRU – use past pattern of block references as a predictor of future references
  - LRU can be bad for some queries
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- Example of bad access pattern for LRU: when computing the join of 2 relations r and s by a nested loops

```
for each tuple tr of r do
  for each tuple ts of s do
    if the tuples tr and ts match …
```

# Buffer-Replacement Policies (Cont.)

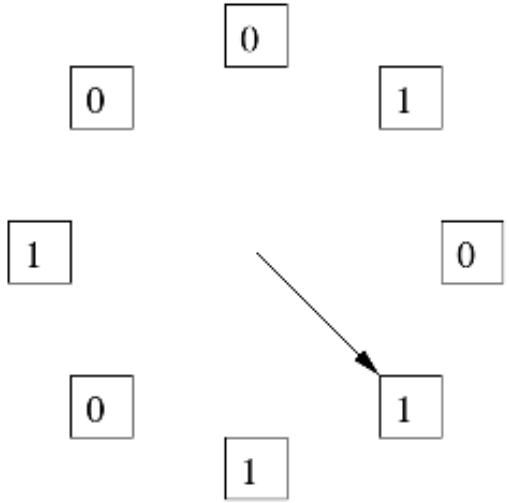
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
  - Can lead to corruption of data structures on disk
    - E.g., linked list of blocks with missing block on disk
    - File systems perform consistency check to detect such situations
  - Careful ordering of writes can avoid many such problems

# Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
  - There are a lot of possible policies.
  - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm.
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
  - The information that will not be needed for the longest time.
  - Is the information that has not been accessed for the longest time.

# The “Clock Algorithm”

- LRU is (perceived to be) expensive
  - Maintain timestamp for each block.
  - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
  - Arrange the frames (places blocks can go) into a logical circle like the seconds on a clock face.
  - Each frame is marked 0 or 1.
    - Set to 1 when block added to frame.
    - Or when application accesses a block in frame.
  - Replacement choice
    - Sweep second hand clockwise one frame at a time.
    - If bit is 0, choose for replacement.
    - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
  - If the second hand is currently at 27 seconds.
  - The 28 second tick mark is “the least recently touched mark.”



# Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- "Scans" are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
  - The engine knows the current position in the result set.
  - Uses the sort order to determine which records will be accessed soon.
  - Tags those blocks as not replaceable.
  - (A form of clairvoyance).
- Not all users/applications are equally "important."
  - Classify users/applications into priority 1, 2 and 3.
  - Sub-allocate the buffer pool into pools P1, P2 and P3.
  - Apply LRU within pools and adjust pool sizes based on relative importance.
  - This prevents
    - A high access rate, low-priority application from taking up a lot of frames
    - Result in low access, high priority applications not getting buffer hits.

# Optimization of Disk Block Access (Cont.)

- Buffer managers support **forced output** of blocks for the purpose of recovery (more in Chapter 19)
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM or flash buffer immediately
  - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
  - Used exactly like nonvolatile RAM
    - Write to log disk is very fast since no seeks are required
- **Journaling file systems** write data in-order to NV-RAM or log disk
  - Reordering without journaling: risk of corruption of file system data

# *Indexes*
# *(Database Systems Concepts, V7, Ch. 14)*

# Outline

- Basic Concepts
- Ordered Indices
- $B^+$-Tree Index Files
- B-Tree Index Files
- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing

# 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

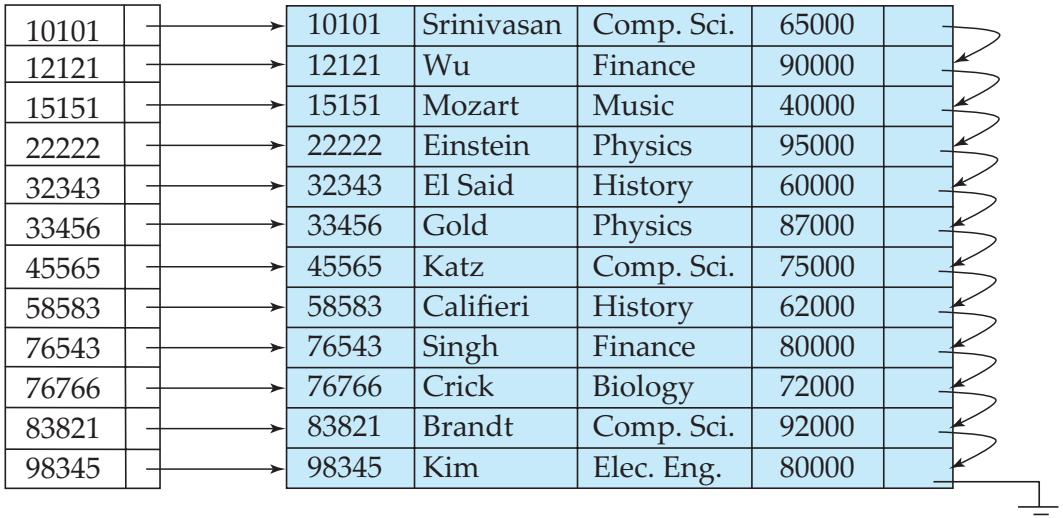
- Access types supported efficiently.  E.g.,
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

# Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary 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 **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.

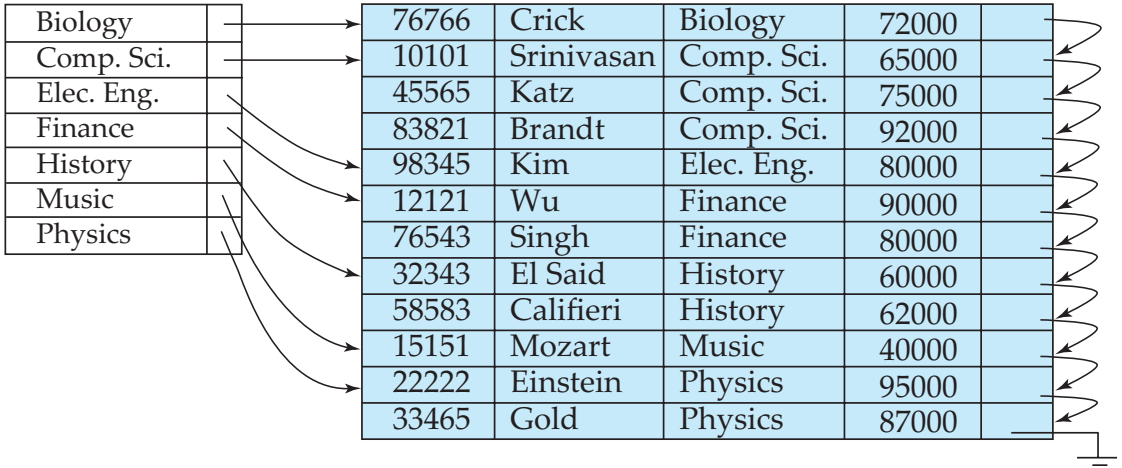
# Dense Index Files

- **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.)

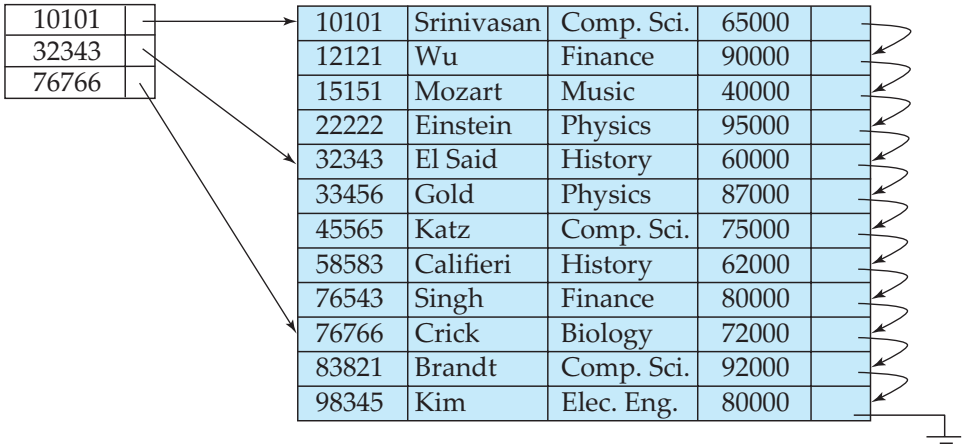
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



# Sparse Index Files

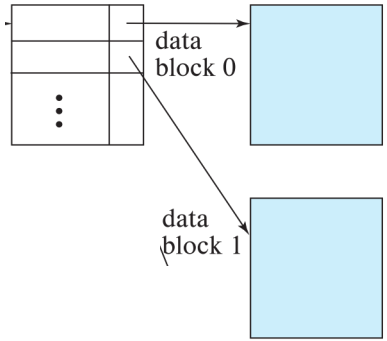
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record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



# Sparse Index Files (Cont.)

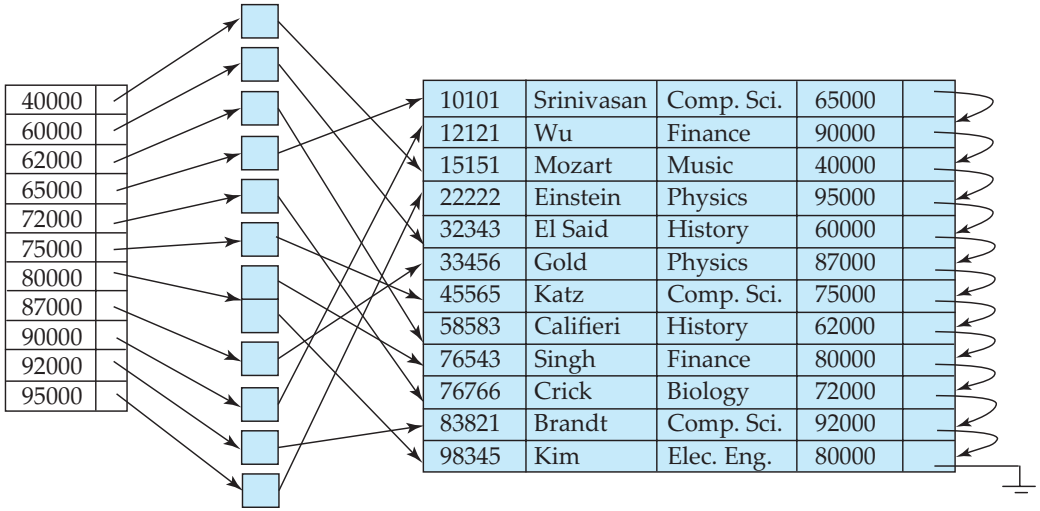
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key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




  - For unclustered index: sparse index on top of dense index (multilevel index)

# 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

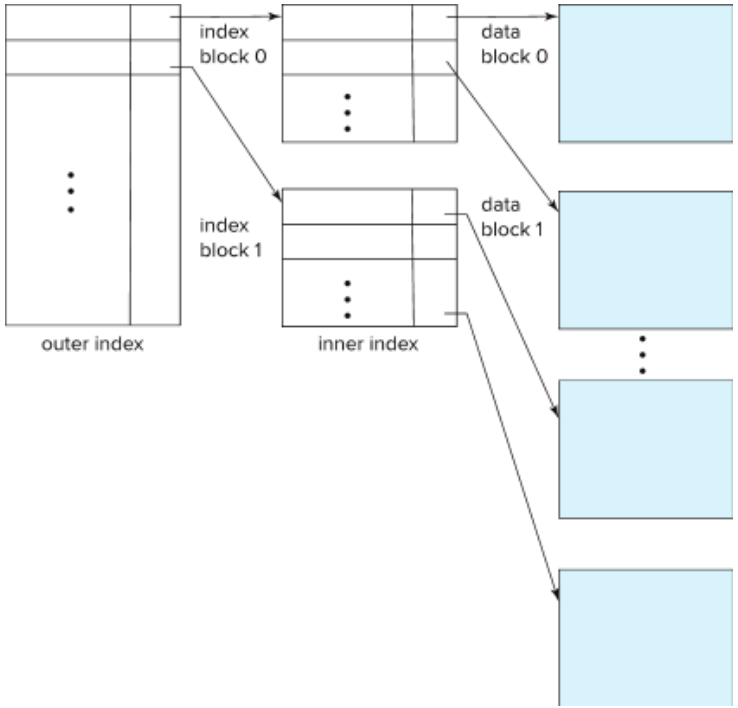
# Clustering vs Nonclustering Indices

- Indices offer substantial benefits when searching for records.
- BUT: indices imposes overhead on database modification
  - when a record is inserted or deleted, every index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated
- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (nonclustering) index is expensive on magnetic disk
  - Each record access may fetch a new block from disk
  - Each block fetch on magnetic disk requires about 5 to 10 milliseconds

# Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic 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.

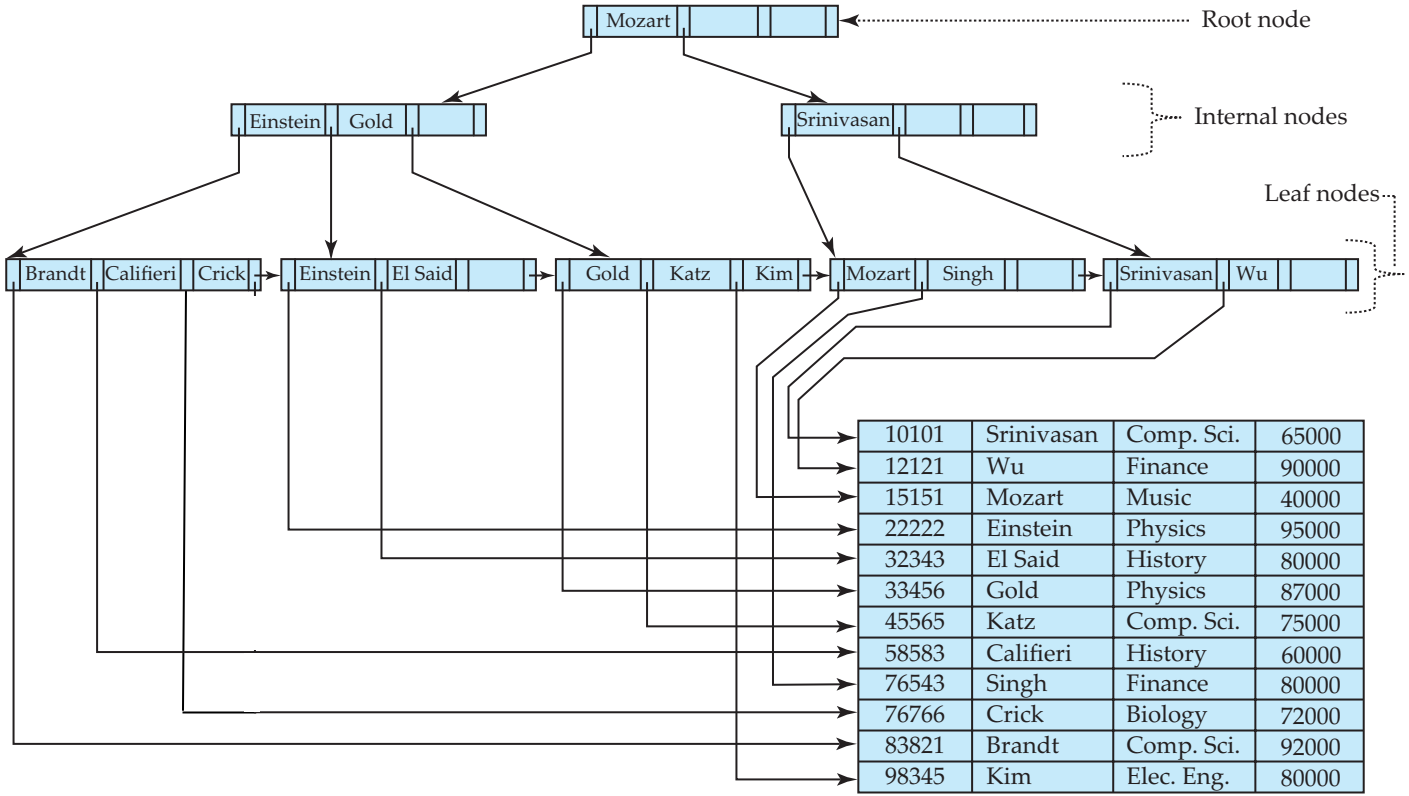
# Multilevel Index (Cont.)



# Indices on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and
      (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

# Example of B+-Tree



# $B^+$-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

| $P_1$ | $K_1$ | $P_2$ | … | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|---|---|---|---|---|---|---|

  - $K_i$ 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

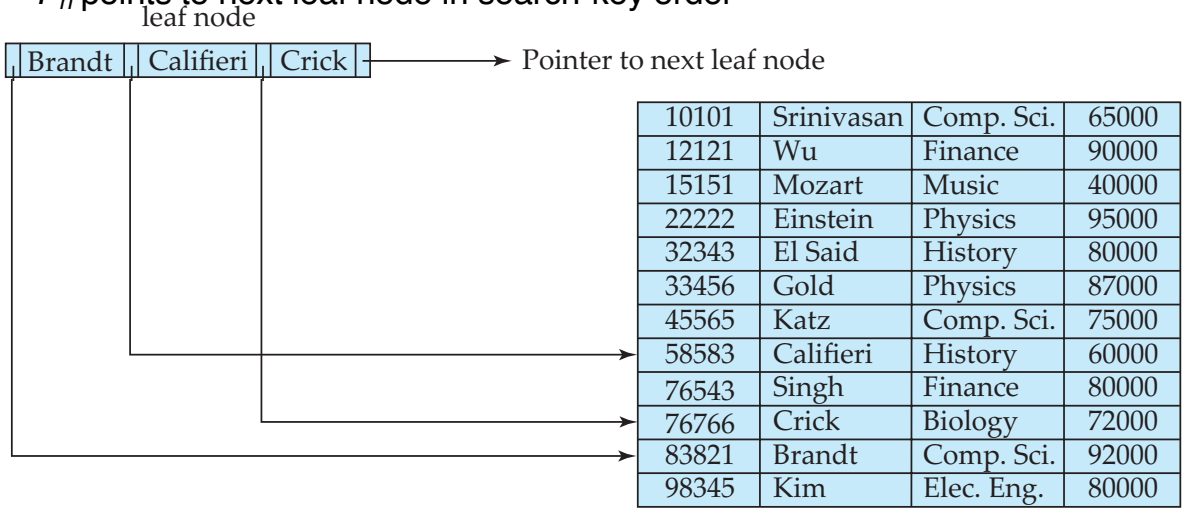
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(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 |

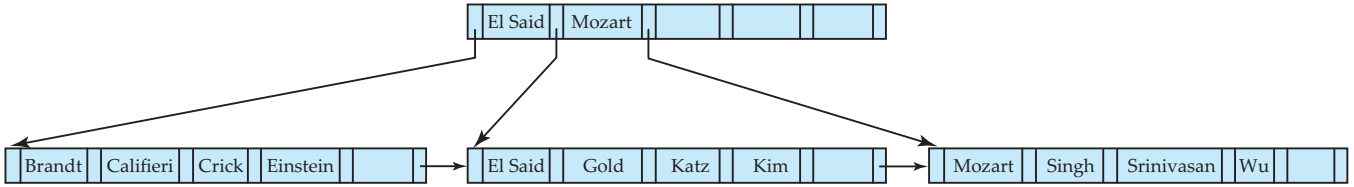
# Non-Leaf Nodes in B+-Trees

- 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 $K_{n-1}$
  - General structure

| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | $\cdots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

# 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.

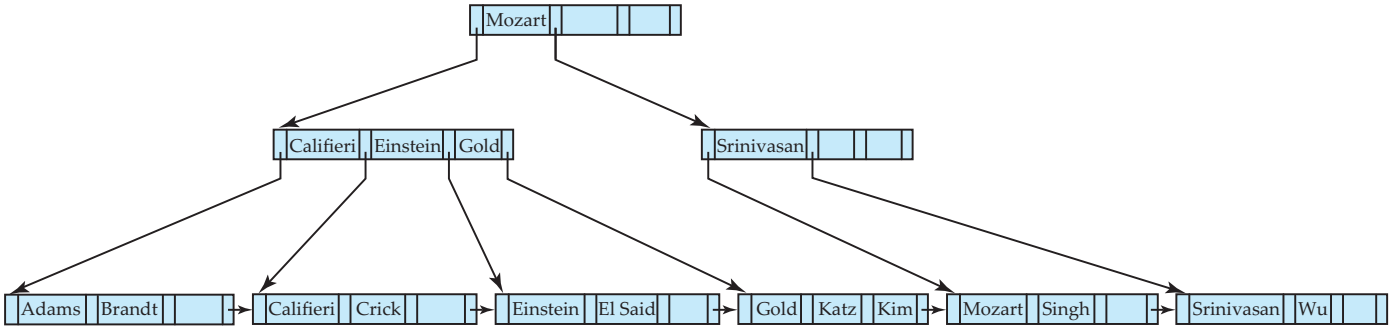
# Observations about $B^+$-trees

- 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

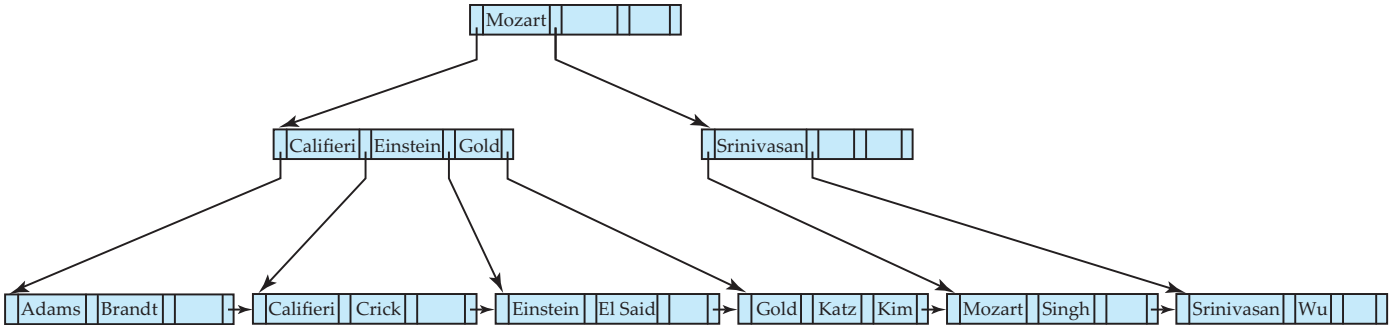
**function** *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i then*
   3. Set *C* = *last non-null pointer in C*
   4. **else if** ($v$ = C.$K_i$ ) Set C = $P_{i+1}$
   5. **else set** *C* = C.$P_i$
3. **if** for some *i*, $K_i = V$ **then** return C.$P_i$
4. **else** return null /* no record with search-key value *v* exists. */



# Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb, ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



# 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 traversal from root to leaf.
- 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

# Non-Unique Keys

- If a search key $a_i$ is not unique, create instead an index on a composite key ($a_i$, $A_p$), which is unique
  - $A_p$ could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range ($v$, $-\infty$) to ($v$, $+\infty$)
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

# Updates on B+-Trees: Insertion

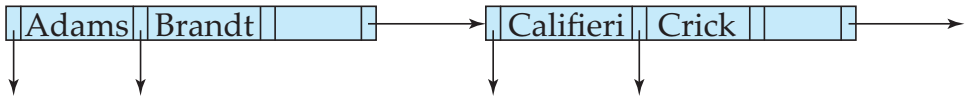
Assume record already added to the file. Let

- *pr* be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
    1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
    2. Otherwise, split the node (along with the new (*v, pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

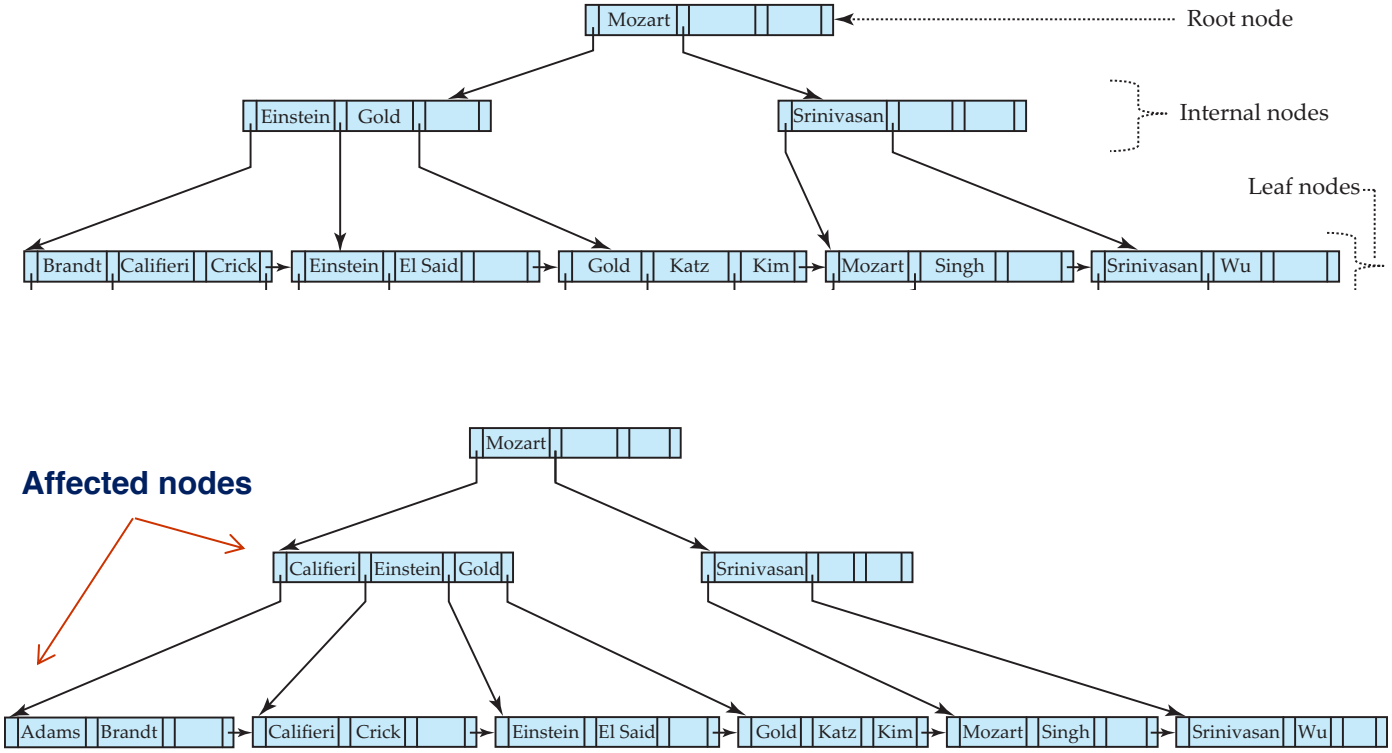
# Updates on B+-Trees: Insertion (Cont.)

- 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.



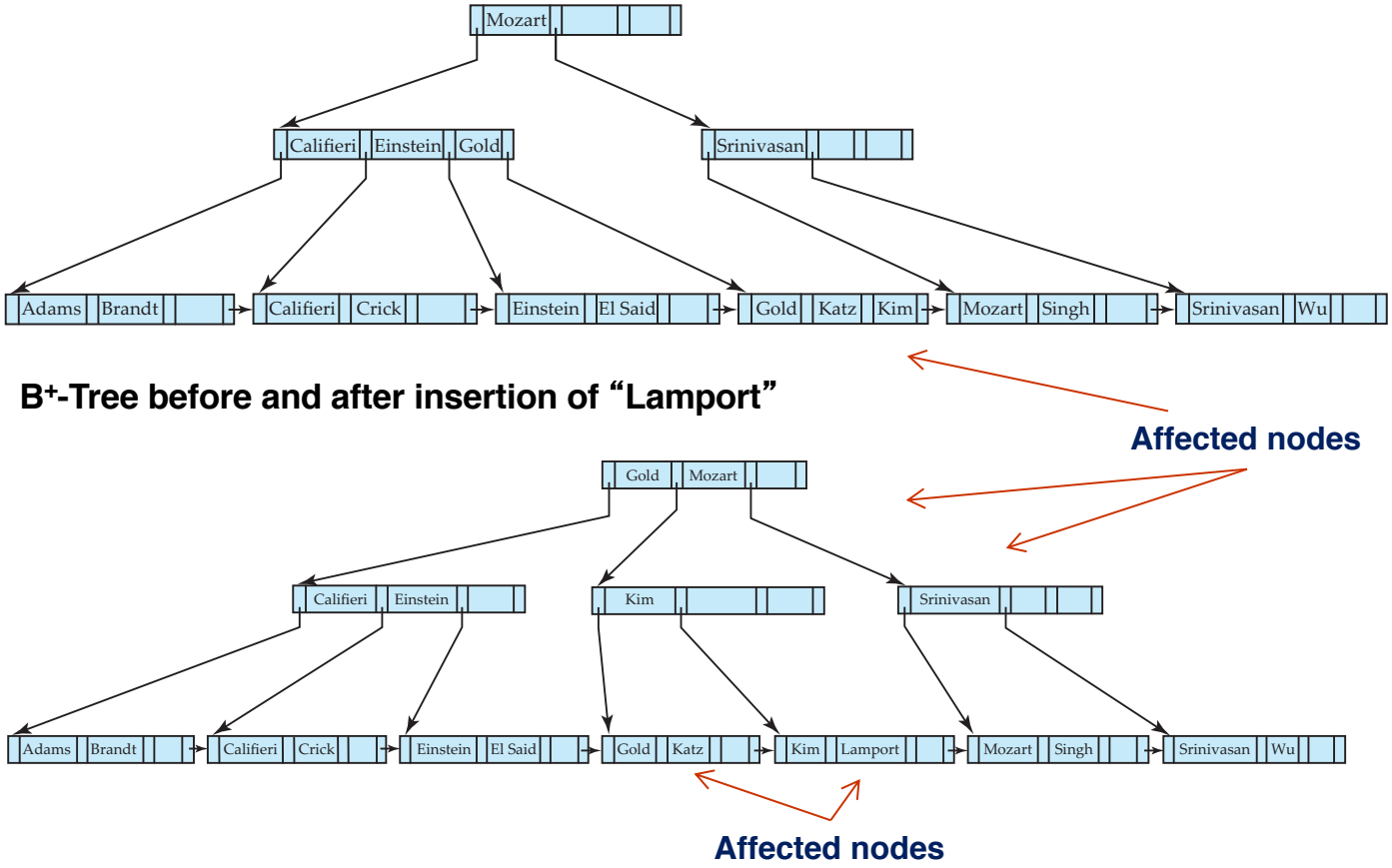

Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

# B+-Tree Insertion



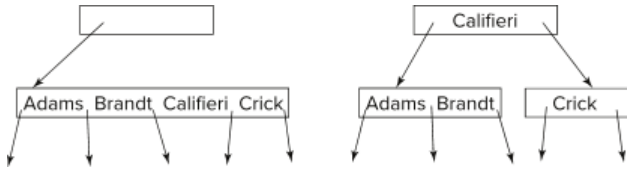

B+-Tree before and after insertion of "Adams"

# B+-Tree Insertion



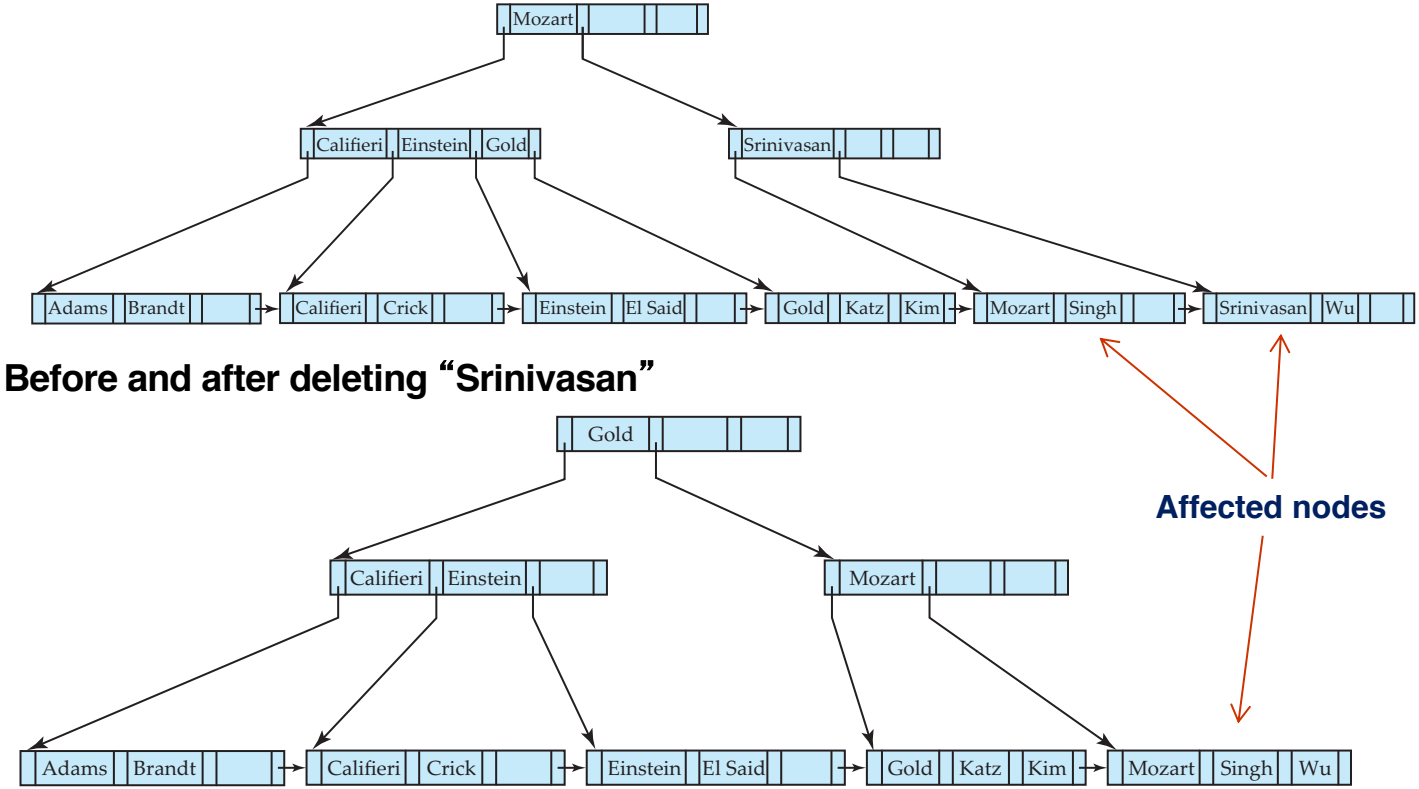
# Insertion in B+-Trees (Cont.)

- 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
- Example



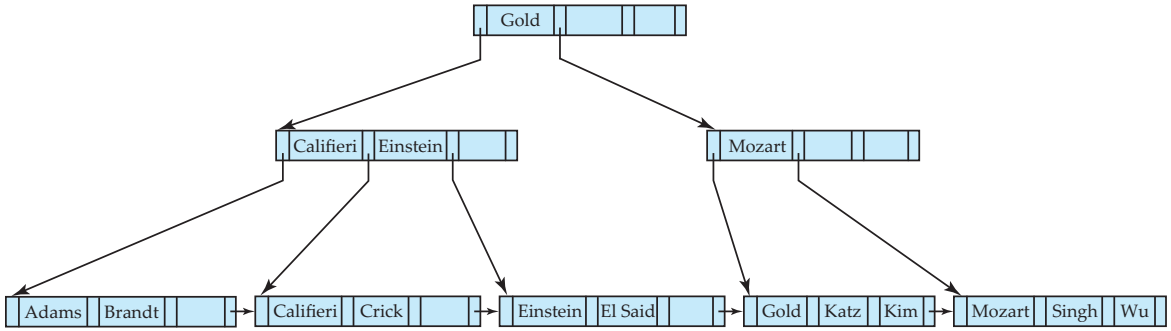

- **Read pseudocode in book!**

# Examples of B+-Tree Deletion

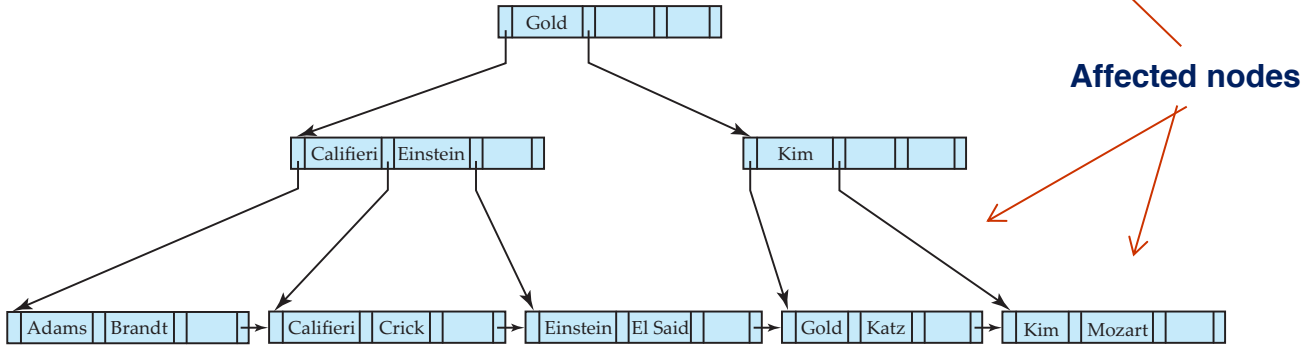



- Deleting "Srinivasan" causes **merging** of under-full leaves

# Examples of B+-Tree Deletion (Cont.)

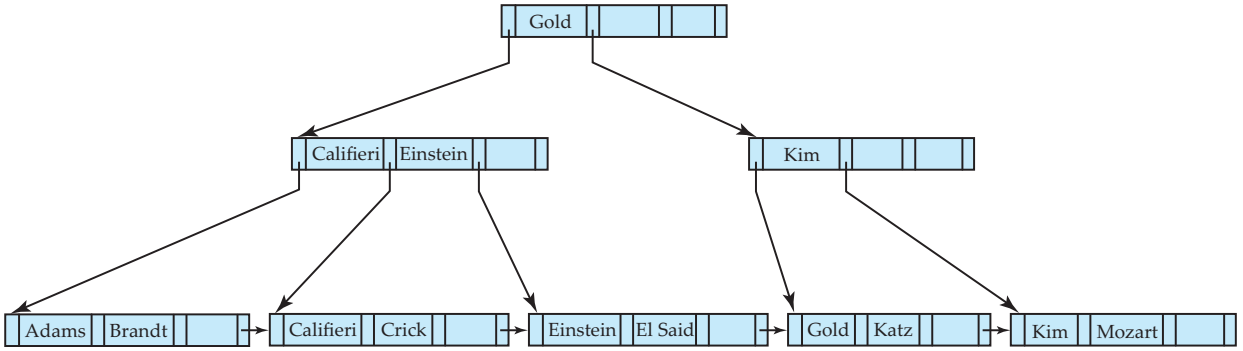



**Before and after deleting "Singh" and "Wu"**

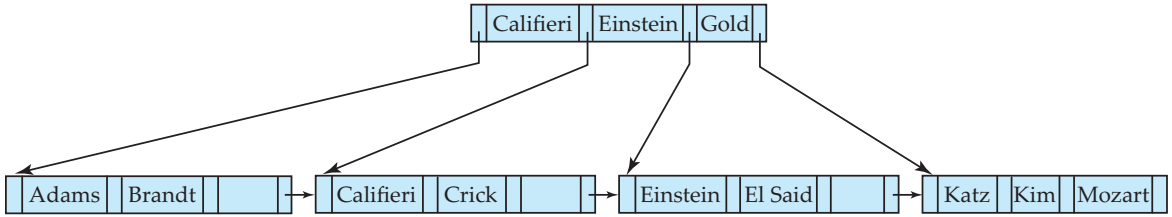



- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

# Example of B+-tree Deletion (Cont.)




**Before and after deletion of "Gold"**




- 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

Assume record already deleted from file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- Remove (*Pr, V*) from the leaf node
- 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.

# Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

# Non-Unique Search Keys

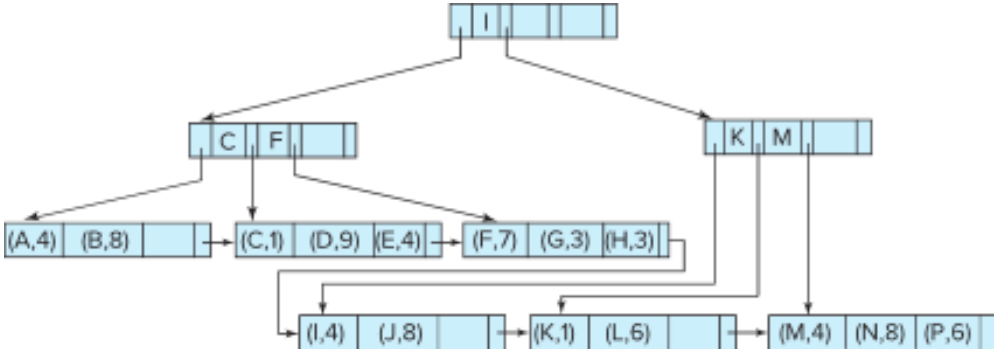
- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
      - Worst case complexity may be linear!
    - 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

- B+-Tree File Organization:
  - Leaf nodes in a B+-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- 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 search key of $B^+$-tree file organization instead of record pointer in secondary index
    - Add record-id if $B^+$-tree file organization search key is non-unique
    - Extra traversal of file organization to locate record
      - Higher cost for queries, but node splits are cheap

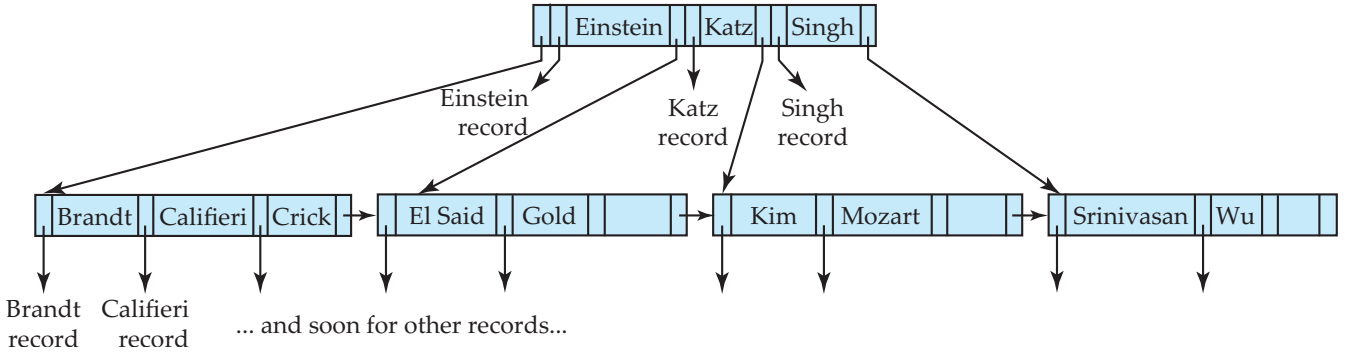
# Indexing Strings

- 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

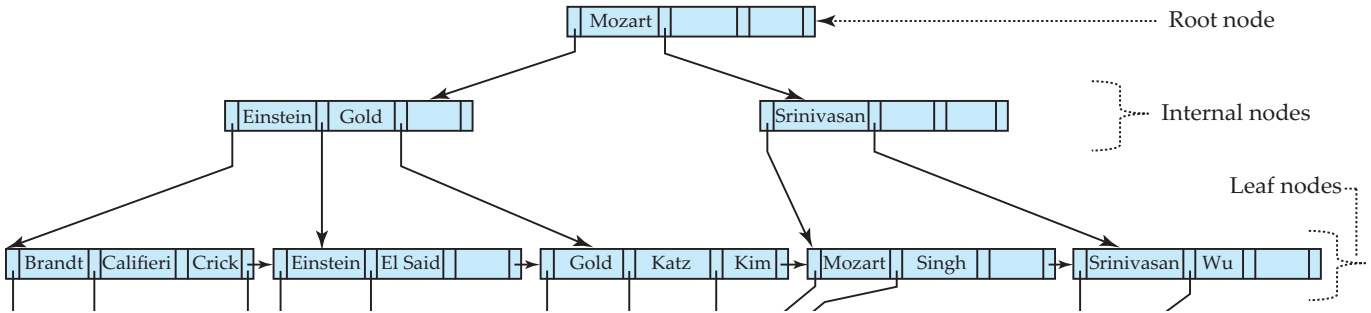
# Bulk Loading and Bottom-Up Build

- 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

# B-Tree Index File Example




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



# Indexing on Flash

- Random I/O cost much lower on flash
  - 20 to 100 microseconds for read/write
- Writes are not in-place, and (eventually) require a more expensive erase
- Optimum page size therefore much smaller
- Bulk-loading still useful since it minimizes page erases
- Write-optimized tree structures (discussed later) have been adapted to minimize page writes for flash-optimized search trees

# Indexing in Main Memory

- Random access in memory
  - Much cheaper than on disk/flash
  - But still expensive compared to cache read
  - Data structures that make best use of cache preferable
  - Binary search for a key value within a large $B^+$-tree node results in many cache misses
- $B^+$- trees with small nodes that fit in cache line are preferable to reduce cache misses
- Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array.

# Hashing

# Static Hashing

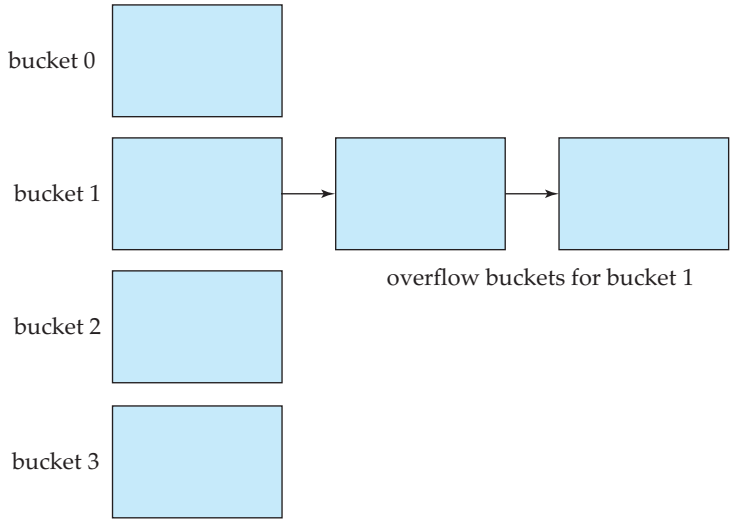
- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

# Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing (**also called **closed hashing or open hashing** depending on the book you use**)**
  - An alternative, called **open addressing (**also called **open hashing** or **closed hashing** depending on the book you use) which does not use over-flow buckets, is not suitable for database applications.



# Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

bucket 0

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

bucket 1

| | | | |
|---|---|---|---|
| 15151 | Mozart | Music | 40000 |
| | | | |
| | | | |
| | | | |

bucket 2

| | | | |
|---|---|---|---|
| 32343 | El Said | History | 80000 |
| 58583 | Califieri | History | 60000 |
| | | | |
| | | | |

bucket 3

| | | | |
|---|---|---|---|
| 22222 | Einstein | Physics | 95000 |
| 33456 | Gold | Physics | 87000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| | | | |

bucket 4

| | | | |
|---|---|---|---|
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| | | | |
| | | | |

bucket 5

| | | | |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| | | | |
| | | | |
| | | | |

bucket 6

| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| | | | |

bucket 7

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

# Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

# Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets

# Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only $B^+$-trees

# Multiple-Key Access

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

  **select** *ID*

  **from** *instructor*

  **where** *dept_name* = “Finance” **and** *salary* = 80000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
  3. Use *dept_name* index to find pointers to all records pertaining to the “Finance” department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

# Indices on Multiple Keys

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

# Indices on Multiple Attributes

Suppose we have an index on combined search-key
(*dept_name, salary*).

- With the **where** clause
  **where** *dept_name* = "Finance" **and** *salary* = 80000
  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  **where** *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  **where** *dept_name* < "Finance" **and** *balance* = 80000
  - May fetch many records that satisfy the first but not the second condition

# Other Features

- **Covering indices**
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
  - Store extra attributes only at leaf
    - Why?
- Particularly useful for secondary indices
  - Why?

# Creation of Indices

- Example
  **create index** *takes_pk* **on** *takes* (*ID,course_ID, year, semester, section*)
  **drop index** *takes_pk*
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
  - Why?
- Some database also create indices on foreign key attributes
  - Why might such an index be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

# Index Definition in SQL

- Create an index

  **create index** <index-name> **on** <relation-name> (<attribute-list>)

  E.g.,: **create index** *b-index* **on** *branch(branch_name)*

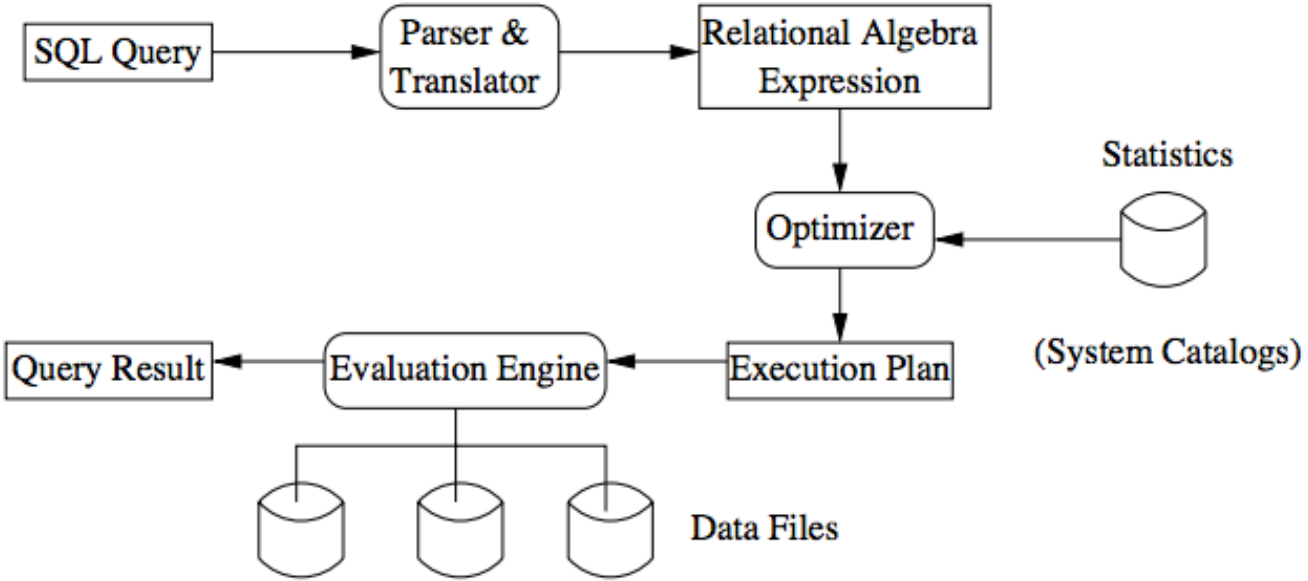
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  **drop index** <index-name>

- Most database systems allow specification of type of index, and clustering.

*Query Processing and Optimization*
*(Database Systems Concepts, V7, Ch. 15)*

## Basic Steps in Processing an SQL Query

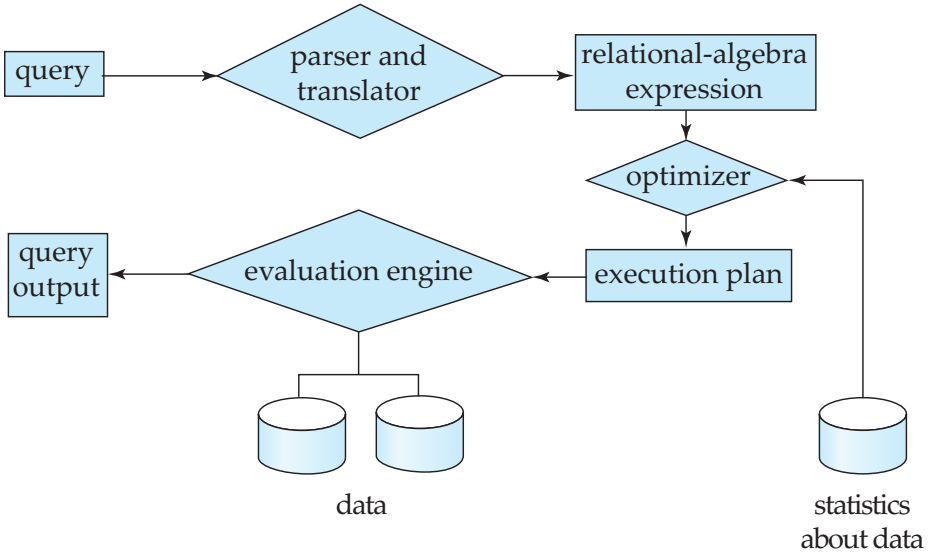


# Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



# Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

# Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- 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.,:
  - Use an index on *salary* to find instructors with salary < 75000,
  - Or perform complete relation scan and discard instructors with salary ≥ 75000

# 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.
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 16
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

# Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk

# Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- 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
    - Assuming for simplicity that write cost is same as read cost
  - $t_S$ – time for one seek
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- $t_S$ and $t_T$ depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: $t_S$ = 4 msec and $t_T$ =0.1 msec
  - SSD: $t_S$ = 20-90 microsec and $t_T$ = 2-10 microsec for 4KB

# Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice

# Selection Operation

- **File scan**
- 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
    - cost = ($b_r$/2) block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

# Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2** (**clustering index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3** (**clustering index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

# Selections Using Indices

- **A4** (**secondary index, equality on key/non-key**).
  - 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!

# Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**clustering index, comparison**). (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A6** (**clustering index, comparison**).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ 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 per record; Linear file scan may be cheaper!

# Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of $\theta_i$ 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.
- **A8** (**conjunctive selection using composite index**).
  - Use appropriate composite (multiple-key) index if available.
- **A9** (**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:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable 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 $\theta$
    - Find satisfying records using index and fetch from file

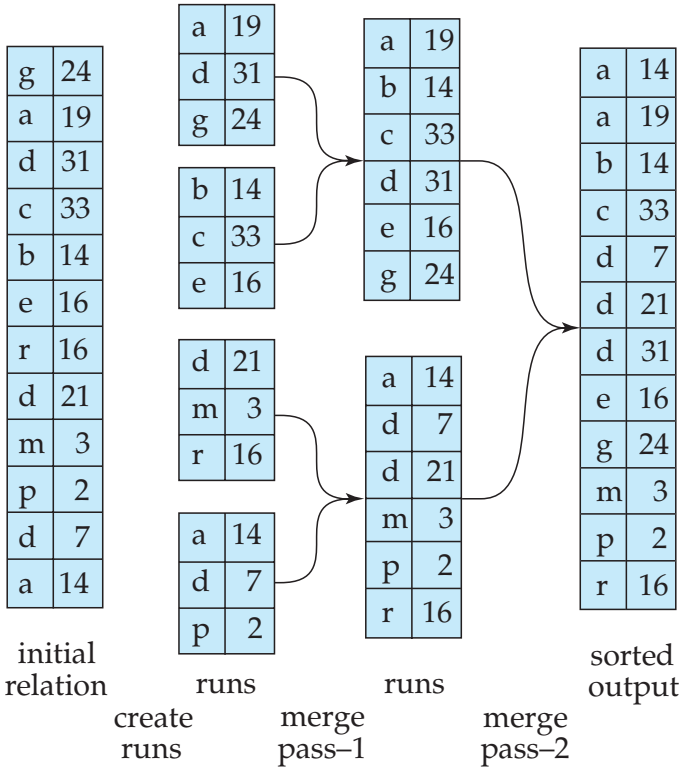
# Bitmap Index Scan

- The **bitmap index scan** algorithm of PostgreSQL
  - Bridges gap between secondary index scan and linear file scan when number of matching records is not known before execution
  - Bitmap with 1 bit per page in relation
  - Steps:
    - Index scan used to find record ids, and set bit of corresponding page in bitmap
    - Linear file scan fetching only pages with bit set to 1
  - Performance
    - Similar to index scan when only a few bits are set
    - Similar to linear file scan when most bits are set
    - Never behaves very badly compared to best alternative

# 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.

# Example: External Sorting Using Sort-Merge



# External Sort-Merge

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
   (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 $N \geq M$, 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.

# External Merge Sort (Cont.)

- Cost analysis:
  - 1 block per run leads to too many seeks during merge
    - Instead use $b_b$ buffer blocks per run
      - ➔ read/write $b_b$ blocks at a time
    - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$.
  - 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:
      $$b_r \left( 2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil + 1 \right) \lceil$$
  - Seeks: next slide

# External Merge Sort (Cont.)

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

# 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 *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

# 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 $\theta$
      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.

# 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

  $n_r * b_s + b_r$ 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
- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

# Block Nested-Loop Join

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

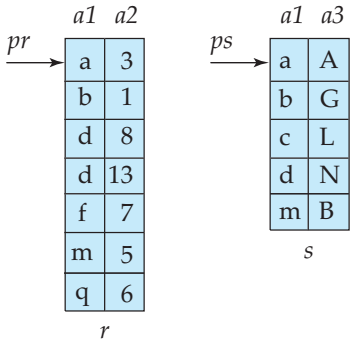
```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r,t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
 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

# 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.*
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
  - 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.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

# 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



# Merge-Join (Cont.)

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

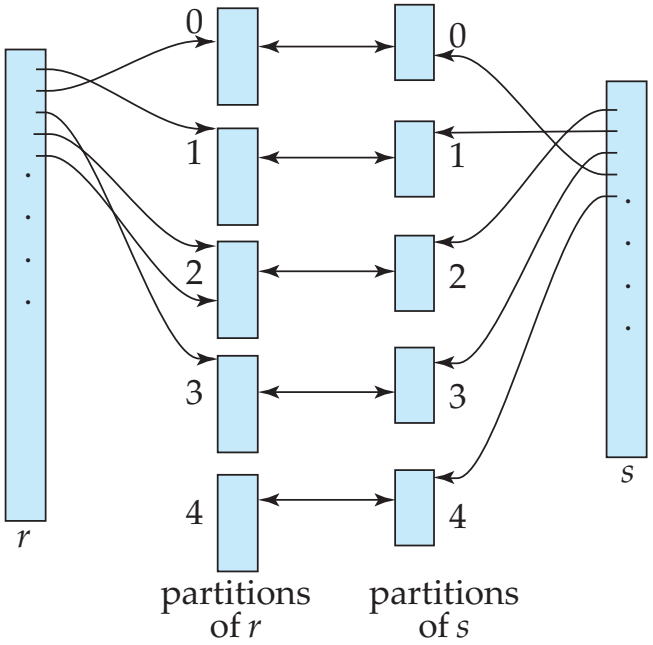
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

  + 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

- 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 $i = h(t_r[JoinAttrs])$.
  - $r_0, r_1 \ldots, r_n$ denotes partitions of $s$ tuples
    - Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[JoinAttrs])$.
- *Note:* In book, Figure 12.10 $r_i$ is denoted as $H_{ri}$, $s_i$ is denoted as $H_{si}$ and $n$ is denoted as $n_h$.

# 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 value $n$ and 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
  - The probe relation partitions $s_i$ 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., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

# Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition $s_i$ if $s_i$ does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition $s_i$ is further partitioned using different hash function.
  - Partition $r_i$ must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g., partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

# Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers +
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation $s$ to less than M blocks per partition is $\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    $2(b_r + b_s)\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +
    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

# Hybrid Hash–Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

  **Keep the first partition of the build relation in memory.**
- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
  - Division of memory:
    - The first partition occupies 20 blocks of memory
    - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
  - the first is used right away for probing, instead of being written out
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M >> \sqrt{b_s}$

# 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 $r \bowtie_{\theta_i} s$:

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

# Joins over Spatial Data

- No simple sort order for spatial joins
- Indexed nested loops join with spatial indices
  - R-trees, quad-trees, k-d-B-trees

# 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 a manner similar 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: **partial aggregation**
    - 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 partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end

# Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ 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
        - $r \cup s$:
          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.

# Other Operations : Set Operations

- E.g., Set operations using hashing:
  1. as before partition *r* and *s,*
  2. as before, process each partition *i* as follows
     1. build a hash index on $r_i$
     2. Process $s_i$ as follows
        - $r \cap s$:
          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.

# Answering Keyword Queries

- Indices mapping keywords to documents
  - For each keyword, store sorted list of document IDs that contain the keyword
    - Commonly referred to as a **inverted index**
    - E.g.,: database: d1, d4, d11, d45, d77, d123
      distributed: d4, d8, d11, d56, d77, d121, d333
  - To answer a query with several keywords, compute intersection of lists corresponding to those keywords
- To support ranking, inverted lists store extra information
  - "**Term frequency**" of the keyword in the document
  - "**Inverse document frequency**" of the keyword
  - **Page rank** of the document/web page

# 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 $r \rtimes s$
  - In $r \rtimes s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r \rtimes s$:
    - 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.

# Other Operations : Outer Join

- Modifying hash join to compute $r \bowtie s$
  - If $r$ is probe relation, output non-matching $r$ tuples padded 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

# Evaluation of Expressions

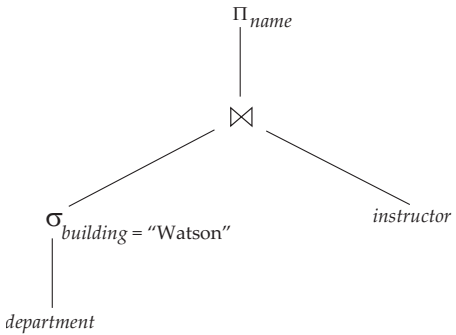
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

# 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

$$\sigma_{building="\text{Watson}"}(department)$$

then compute the store its join with *instructor,* and finally compute the projection on *name.*



# Materialization (Cont.)

- Materialized evaluation is always applicable
- 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

$$\sigma_{building = \text{"Watson"}}(department)$$

  - 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**

# Pipelining (Cont.)

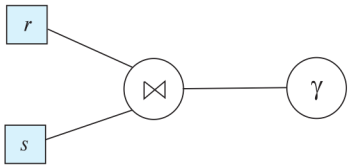
- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** 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
- Alternative name: **pull** and **push** models of pipelining

# Pipelining (Cont.)

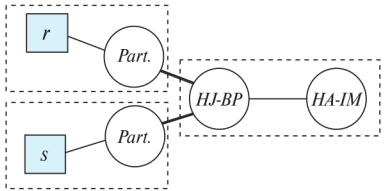
- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g., file scan: initialize file scan
        - state: pointer to beginning of file
      - E.g., merge join: sort relations;
        - state: 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()**

# Blocking Operations

- **Blocking operations**: cannot generate any output until all input is consumed
  - E.g., sorting, aggregation, …
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - E.g., for sort: run generation and merge
  - For hash join: partitioning and build-probe
- Treat them as separate operations



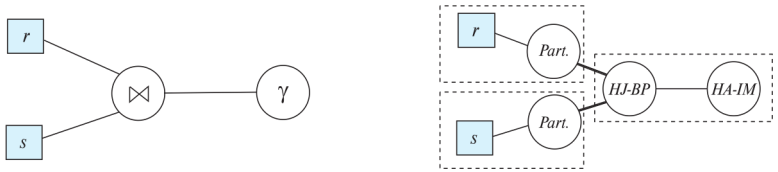

(a) Logical Query




(b) Pipelined Plan

# Pipeline Stages

- **Pipeline stages**:
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution




(a) Logical Query

(b) Pipelined Plan

# 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 generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
  - **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new $r_0$ tuple is found, match it with existing $s_0$ tuples, output matches, and save it in $r_0$
    - Symmetrically for $s_0$ tuples

# Pipelining for Continuous-Stream Data

- **Data streams**
  - Data entering database in a continuous manner
  - E.g., Sensor networks, user clicks, …
- **Continuous queries**
  - Results get updated as streaming data enters the database
  - Aggregation on windows is often used
    - E.g., **tumbling windows** divide time into units, e.g., hours, minutes
- Need to use pipelined processing algorithms
  - **Punctuations** used to infer when all data for a window has been received

# Query Processing in Memory

- Query compilation to machine code
  - Overheads of interpretation
    - E.g., repeatedly finding attribute location within tuple, from metadata
    - Overhead of expression evaluation
  - Compilation can avoid many such overheads and speed up query processing
  - Often via generation of Java byte code / LLVM, with just-in-time (JIT) compilation
- Column-oriented storage
  - Allows vector operations (in conjunction with compilation)
- Cache conscious algorithms

# Cache Conscious Algorithms

- Goal: minimize cache misses, make best use of data fetched into the cache as part of a cache line
- For sorting:
  - Use runs that are as large as L3 cache (a few megabytes) to avoid cache misses during sorting of a run
  - Then merge runs as usual in merge-sort
- For hash-join
  - First create partitions such that build+probe partitions fit in memory
  - Then subpartition further s.t. build subpartition+index fits in L3 cache
    - Speeds up probe phase significantly by avoiding cache misses
- Lay out attributes of tuples to maximize cache usage
  - Attributes that are often accessed together should be stored adjacent to each other
- Use multiple threads for parallel query processing
  - Cache misses leads to stall of one thread, but others can proceed

# End of Chapter 15