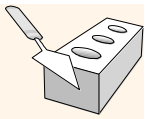
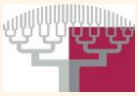
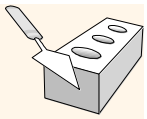


Relational Databases

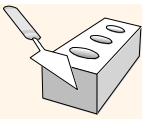


Themis Palpanas
Paris Descartes University

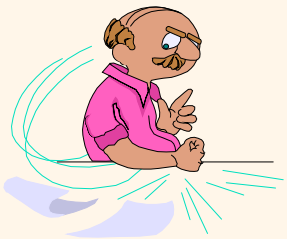


Acknowledgements

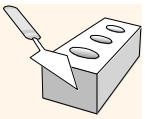
- thanks for slides to
  - Raghu Ramakrishnan
  - Johannes Gehrke
  - Joe Hellerstein
  - Christopher Olston
  - Rui Zhang
  - Brian Lee



Database Management Systems



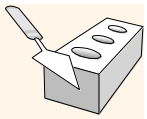
Chapter 1



What Is a DBMS?

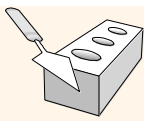
- A very large, integrated collection of data.
- Models real-world *enterprise.*
  - Entities (e.g., students, courses)
  - Relationships (e.g., Madonna is taking CS564)
- A *Database Management System (DBMS)* is a software package designed to store and manage databases.

Files vs. DBMS



- Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)
- Special code for different queries
- Must protect data from inconsistency due to multiple concurrent users
- Crash recovery
- Security and access control

Why Use a DBMS?

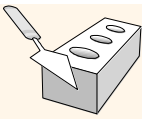


- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.

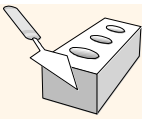
Why Study Databases??

- Shift from *computation* to *information*
  - at the "low end": scramble to webspace (a mess!)
  - at the "high end": scientific applications
- Datasets increasing in diversity and volume.
  - Digital libraries, interactive video, Human Genome project, EOS project
  - ... need for DBMS exploding
- DBMS encompasses most of CS
  - OS, languages, theory, AI, multimedia, logic
- Modern data management solutions based on same/similar ideas

Data Models

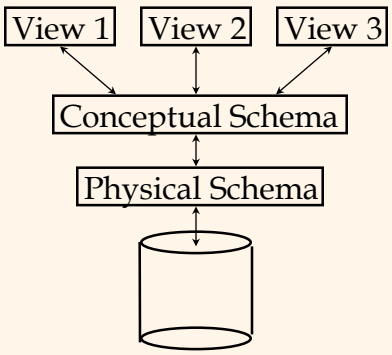


- A *data model* is a collection of concepts for describing data.
- A *schema* is a description of a particular collection of data, using the a given data model.
- The *relational model of data* is the most widely used model today.
  - Main concept: *relation*, basically a table with rows and columns.
  - Every relation has a *schema*, which describes the columns, or fields.

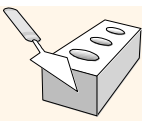


Levels of Abstraction

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



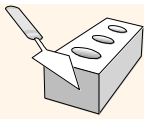
** Schemas are defined using DDL; data is modified/queried using DML.*



Example: University Database

- Conceptual schema:
  - *Students(sid: string, name: string, login: string, age: integer, gpa:real)*
  - *Courses(cid: string, cname:string, credits:integer)*
  - *Enrolled(sid:string, cid:string, grade:string)*
- Physical schema:
  - Relations stored as unordered files.
  - Index on first column of Students.
- External Schema (View):
  - *Course_info(cid:string,enrollment:integer)*

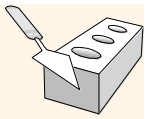
*Data Independence **



- Applications insulated from how data is structured and stored.
- *Logical data independence*: Protection from changes in *logical* structure of data.
  - eg, adding an attribute to a table
- *Physical data independence*: Protection from changes in *physical* structure of data.
  - eg, sorting the file that stores a table

** One of the most important benefits of using a DBMS!*

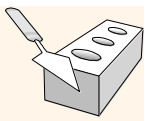
Concurrency Control



- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed.
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

Transaction: An Execution of a DB Program

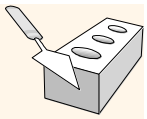
- Key concept is *transaction*, which is an *atomic* sequence of database actions (reads/writes).
- Each transaction, executed completely, must leave the DB in a *consistent state* if DB is consistent when the transaction begins.
  - Users can specify some simple *integrity constraints* on the data, and the DBMS will enforce these constraints.
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
  - Thus, ensuring that a transaction (run alone) preserves consistency is ultimately the user's responsibility!



Scheduling Concurrent Transactions

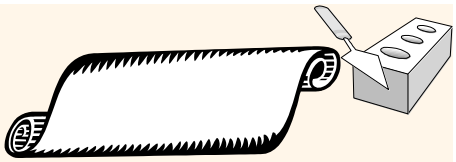
- DBMS ensures that execution of {T1, ... , Tn} is equivalent to some *serial* execution T1' ... Tn'.
  - Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock. All locks are released at the end of the transaction. (Strict 2PL locking protocol.)
  - Idea: If an action of Ti (say, writing X) affects Tj (which perhaps reads X), one of them, say Ti, will obtain the lock on X first and Tj is forced to wait until Ti completes; this effectively orders the transactions.
  - What if Tj already has a lock on Y and Ti later requests a lock on Y? (Deadlock!) Ti or Tj is aborted and restarted!

Ensuring Atomicity



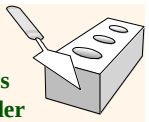
- DBMS ensures *atomicity* (all-or-nothing property) even if system crashes in the middle of a Xact.
- Idea: Keep a *log* (history) of all actions carried out by the DBMS while executing a set of Xacts:
  - Before a change is made to the database, the corresponding log entry is forced to a safe location. (*WAL protocol*; OS support for this is often inadequate.)
  - After a crash, the effects of partially executed transactions are *undone* using the log. (Thanks to WAL, if log entry wasn't saved before the crash, corresponding change was not applied to database!)

The Log



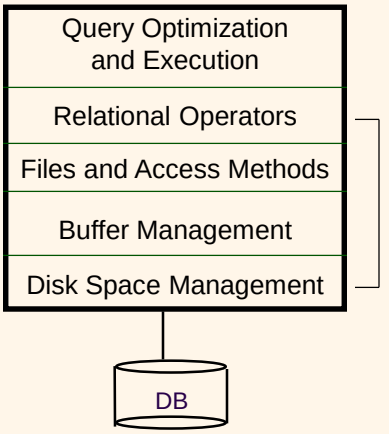
- The following actions are recorded in the log:
  - *Ti writes an object*: The old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: A log record indicating this action.
- Log records chained together by Xact id, so it's easy to undo a specific Xact (e.g., to resolve a deadlock).
- Log is often *duplexed* and *archived* on "stable" storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

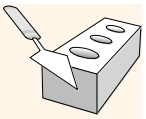
Structure of a DBMS



These layers must consider concurrency control and recovery

- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.

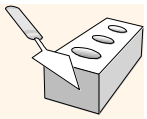




Relational Algebra

Chapter 4, Part A

Relational Query Languages

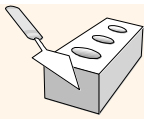


- *Query languages*: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages **!=** programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

Preliminaries



- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Example Instances

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

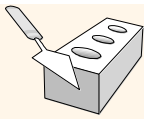
S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

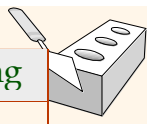
sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Relational Algebra



- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Projection



- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

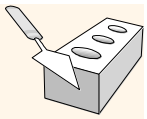
sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

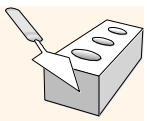


- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

  - *Renaming operator*: $\rho\,(C(1 \to sid1, 5 \to sid2),\ S1 \times R1)$

Joins

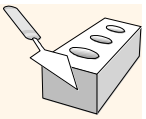


- *Condition Join*: $R \bowtie_c S = \sigma_c (R \times S)$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.



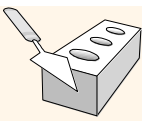
Joins

- *Equi-Join*: A special case of condition join where the condition *c* contains only ***equalities.***

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$$S1 \bowtie_{sid} R1$$

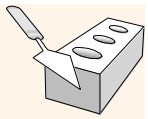
- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields.



Division

- Not supported as a primitive operator, but useful for expressing queries like:
  *Find sailors who have reserved **all** boats.*
- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - $A/B = \{\langle x \rangle \mid \exists \langle x, y \rangle \in A \ \forall \langle y \rangle \in B\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every* *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and $x \cup y$ is the list of fields of *A*.

Examples of Division A/B



sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

A/B2

pno
p1
p2
p4

B3

sno
s1

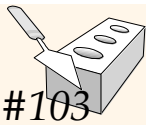
A/B3

Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea*: For *A/B*, compute all *x* values that are not `disqualified' by some *y* value in *B*.
  - *x* value is *disqualified* if by attaching *y* value from *B*, we obtain an *xy* tuple that is not in *A*.

Disqualified *x* values: $\pi_x((\pi_x(A) \times B) - A)$

A/B: $\pi_x(A)$ − all disqualified tuples

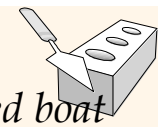


Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103} \text{Reserves}) \bowtie Sailors)$
- Solution 2: $\rho\ (Temp1,\ \sigma_{bid=103} \text{Reserves})$

  $\rho\ (Temp2,\ Temp1 \bowtie Sailors)$

  $\pi_{sname}\ (Temp2)$
- Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \bowtie Sailors))$



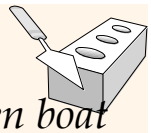
Find names of sailors who've reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

  $\pi_{sname}((\sigma_{color='red'} Boats) \bowtie \text{Reserves} \bowtie Sailors)$
- A more efficient solution:

  $\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'} Boats) \bowtie \text{Res}) \bowtie Sailors)$

A query optimizer can find this, given the first solution!



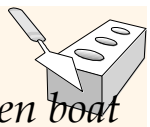
Find sailors who've reserved a red or a green boat

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}, Boats))$$

$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

- Can also define Tempboats using union! (How?)
- What happens if $\vee$ is replaced by $\wedge$ in this query?



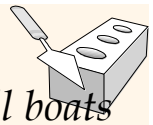
Find sailors who've reserved a red and a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'}, Boats) \bowtie Reserves))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'}, Boats) \bowtie Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$



Find the names of sailors who've reserved all boats

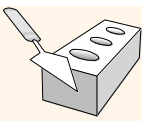
- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\,(Tempsids, (\pi_{sid,bid} \text{Re}serves) / (\pi_{bid} Boats))$$

$$\pi_{sname}(Tempsids \bowtie Sailors)$$

- To find sailors who've reserved all 'Interlake' boats:

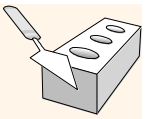
$$..... \; / \; \pi_{bid}(\sigma_{bname='Interlake'} Boats)$$



Storing Data: Disks and Files

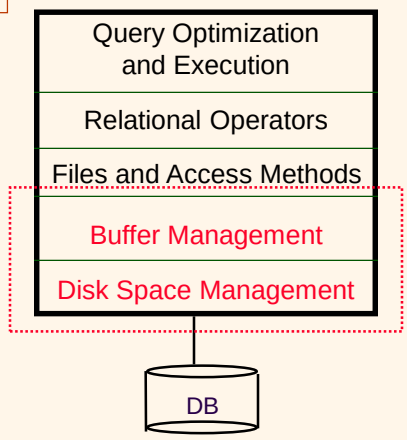
Chapter 9

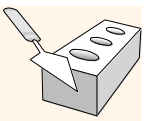
"Yea, from the table of my memory
I'll wipe away all trivial fond records."
-- Shakespeare, *Hamlet*



Disks, Memory, and Files

The BIG picture…

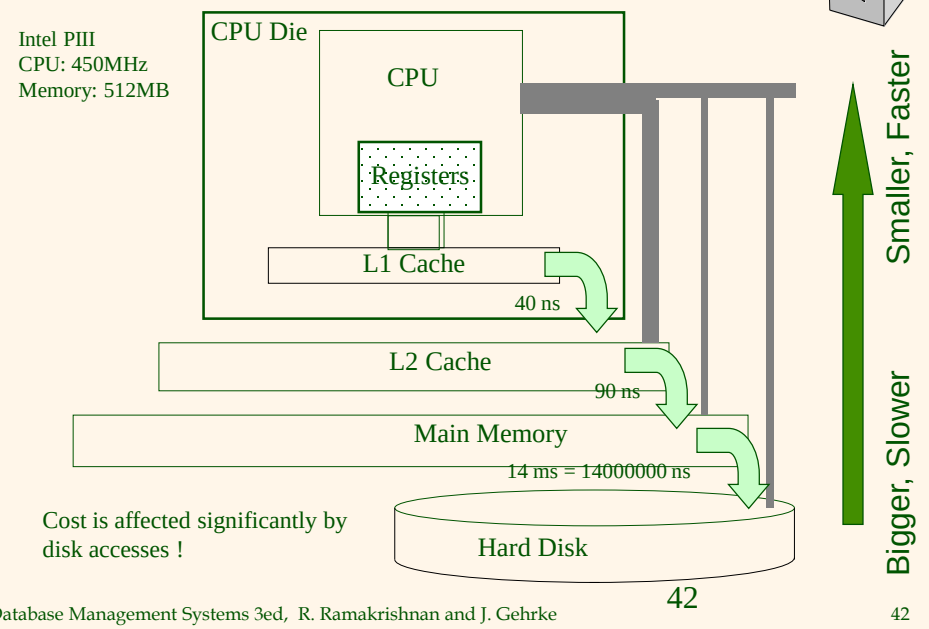




Disks and Files

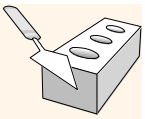
- DBMS stores information on ("hard") disks.
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

The Memory Hierarchy



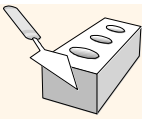
Why Not Store Everything in Main Memory?

- *Costs too much.* $100 will buy you either 16GB of RAM, or 3TB of (spinning) disk (or 128GB SSD) today.
- *Main memory is volatile.* We want data to be saved between runs. (Obviously!)
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).



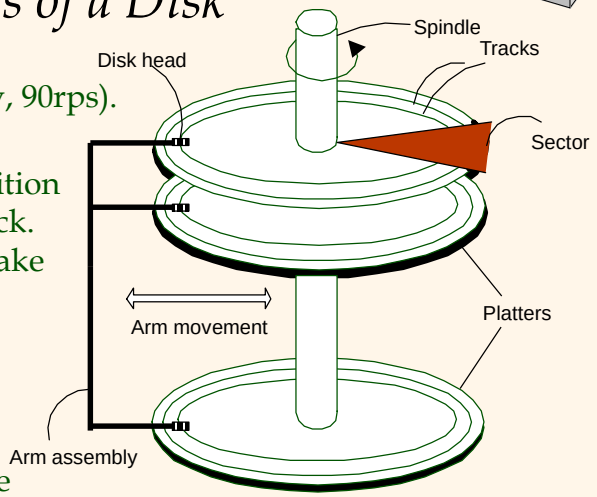
Disks

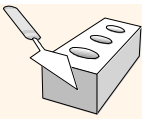
- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!



Components of a Disk

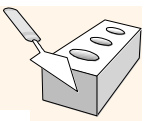
- The platters spin (say, 90rps).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed).



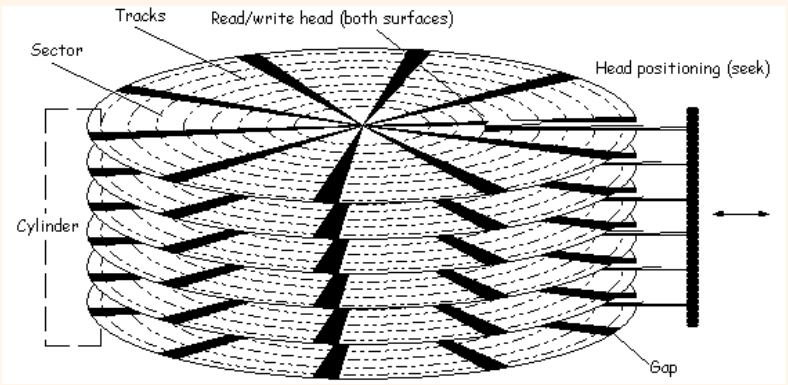


Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 0.3 to 10msec
  - Rotational delay varies from 0 to 4msec
  - Transfer rate is about 0.08msec per 8KB page
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?



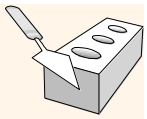
The Hard (Magnetic) disk



- The time for a ***disk block access***, or ***disk page access*** or ***disk I/O***
  access time = seek time + rotational delay + transfer time
- IBM Deskstar 14GPX: 14.4GB
  - Seek time: 9.1 msec
  - Rotational delay: 4.17 msec
  - Transfer rate: 13MB/sec, that is, 0.3msec/4KB

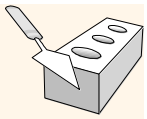
47

Arranging Pages on Disk



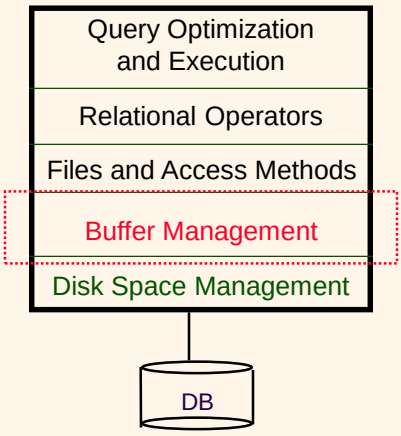
- `*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Disk Space Management

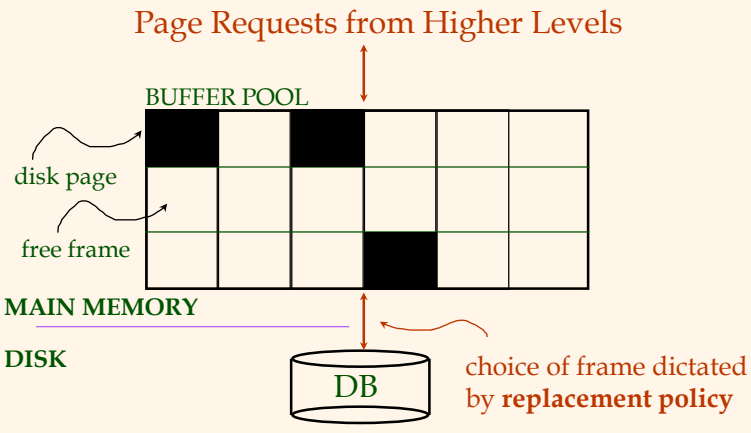


- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed.

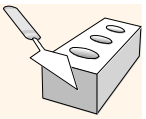
Context



Buffer Management in a DBMS



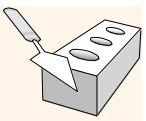
- *Data must be in RAM for DBMS to operate on it!*
- *Buffer Mgr hides the fact that not all data is in RAM*



When a Page is Requested ...

- Buffer pool information table contains: *<frame#, pageid, pin_count, dirty>*
- If requested page is not in pool:
  - Choose a frame for *replacement*
    Only "un-pinned" pages are candidates!
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

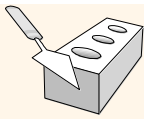
* *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!*



More on Buffer Management

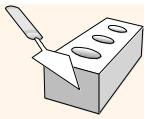
- Requestor of page must eventually unpin it, and indicate whether page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used.
  - To pin a page, pin_count++
  - A page is a candidate for replacement iff *pin count* == 0 ("*unpinned*")
- CC & recovery may entail additional I/O when a frame is chosen for replacement.
  - *Write-Ahead Log* protocol; more later!

Buffer Replacement Policy



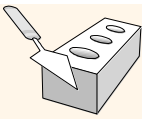
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern*.

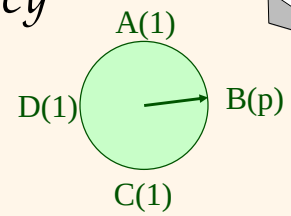
LRU Replacement Policy



- *Least Recently Used (LRU)*
  - for each page in buffer pool, keep track of time when last *unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages
- Problems?
- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Idea: MRU better in this scenario?

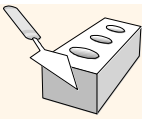
"Clock" Replacement Policy





- An approximation of LRU
- Arrange frames into a cycle, store one *reference bit per frame*
  - Can think of this as the 2nd chance bit
- When pin count reduces to 0, turn on ref. bit
- When replacement necessary

```
do for each page in cycle {
        if (pincount == 0 && ref bit is on)
                turn off ref bit;
        else if (pincount == 0 && ref bit is off)
                choose this page for replacement;
} until a page is chosen;
```

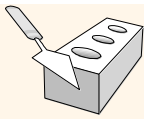



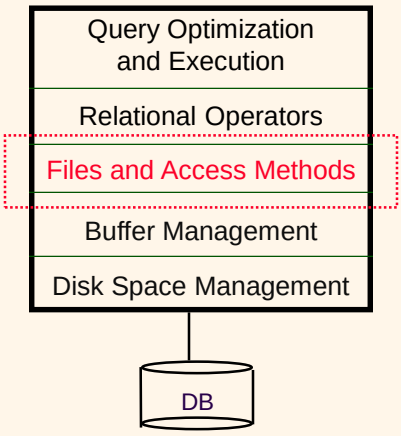
DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

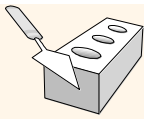
- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

Context

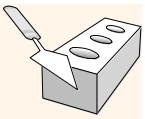




Files of Records

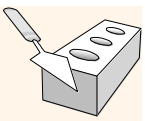


- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

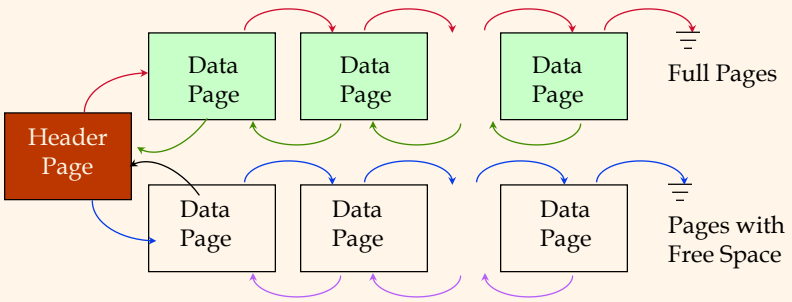


Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.

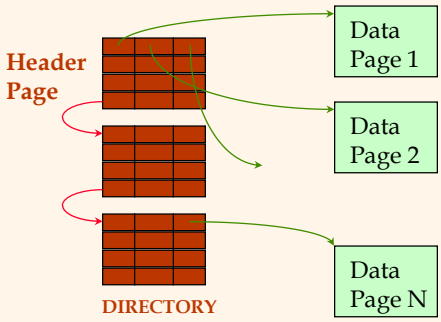


Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers' plus data.

Heap File Using a Page Directory

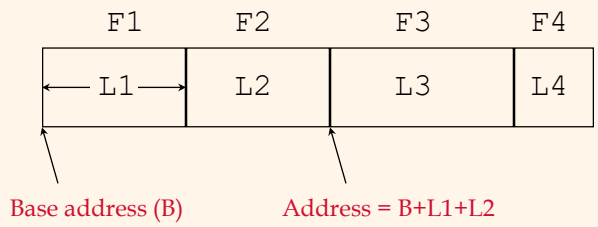


- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages*!

Indexes (a sneak preview)

- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such value-based queries efficiently.

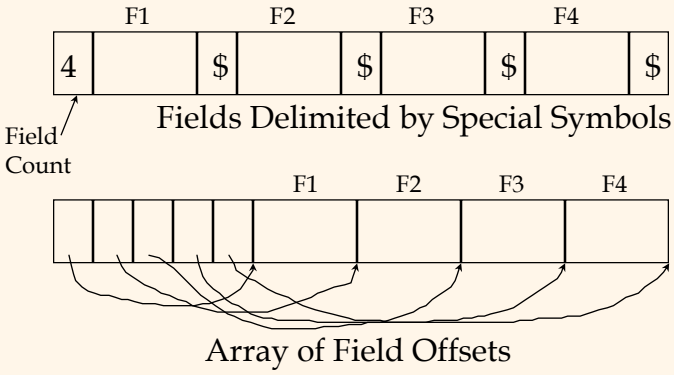
Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field does not require scan of record.

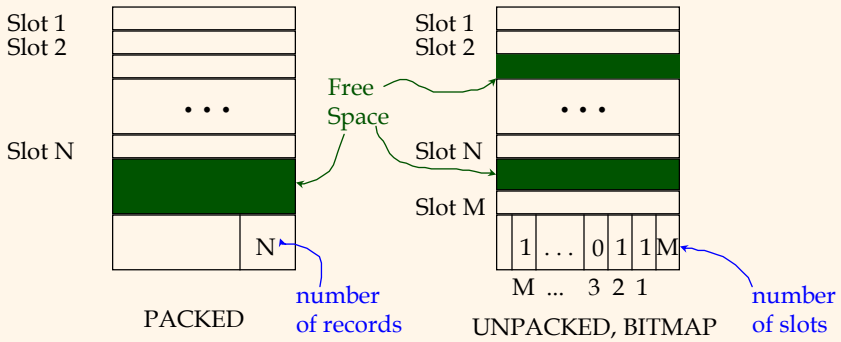
Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



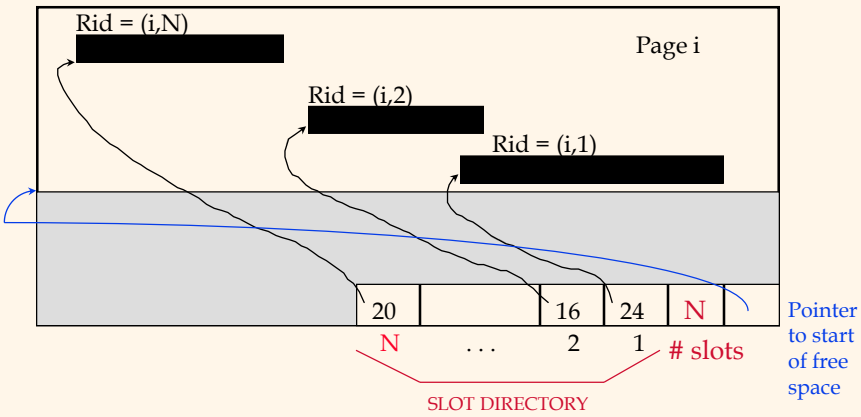
* Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Page Formats: Fixed Length Records



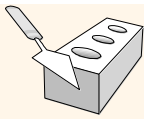
* *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records



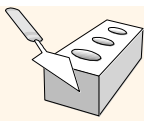
* *Can move records on page without changing rid; so, attractive for fixed-length records too.*

Data on External Storage



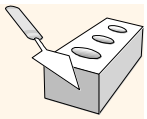
- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage
- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Alternative File Organizations



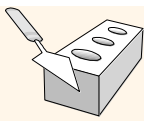
Many alternatives exist, *each ideal for some situations, and not so good in others:*

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.



Tree-Structured Indexes

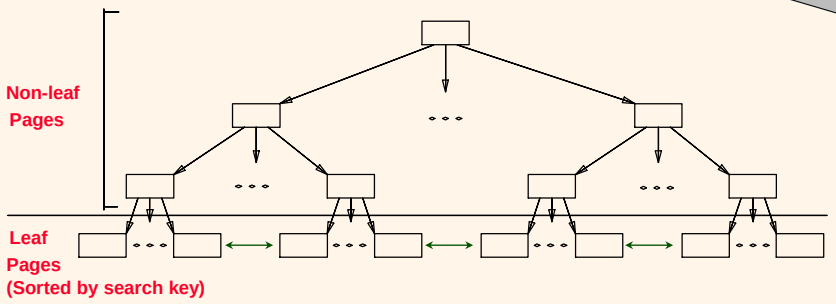
Chapter 10



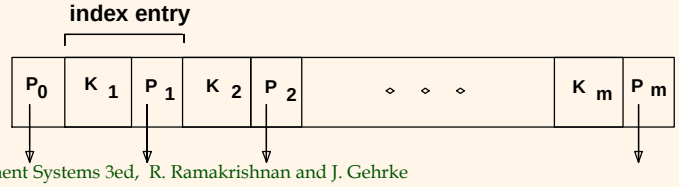
Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

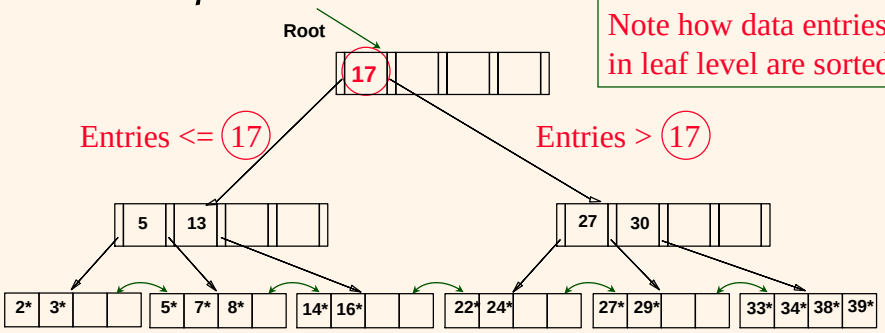
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



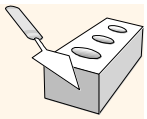
Example B+ Tree



Note how data entries in leaf level are sorted

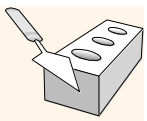
- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

History of B+ Trees



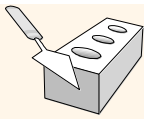
- First described in paper by Rudolf Bayer and Edward M. McCreight in 1972
- "Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189 (1972)"

Description of B+ Trees



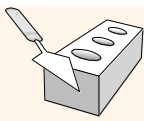
- A variation of B-Trees
- B-Trees commonly found in databases and filesystems
- Sacrifices space for efficiency (not as much re-balancing required compared to other balanced trees)

Description of B+ Trees (cont.)



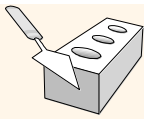
- Main difference of B-Trees and B+ Trees
  - B-Trees: data stored at every level (in every node)
  - B+ Trees: data stored only in leaves
    - Internal nodes only contain keys and pointers
    - All leaves are at the same level (the lowest one)
    - All leaves are linked in a chain via pointers from one leaf to the next

Description of B+ Trees (cont.)



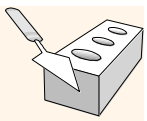
- B+ trees have an order d
- An internal node can have up to d-1 keys and d pointers (children)
- Built from the bottom up

Description of B+ Trees (cont.)



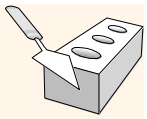
- All nodes must have between ceil(d/2) and n keys (except for the root, which can have at least 1 key and 2 pointers)
- For a B+ tree of order d and height h, it can hold up to d^h-d^{h-1} keys

Searching in B+ Trees



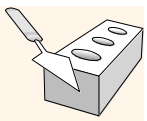
- Searching just like in a binary search tree
- Starts at the root, works down to the leaf level
- Does a comparison of the search value and the current "separation value", goes left or right

Inserting into a B+ Tree



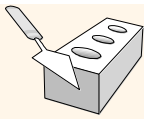
- A search is first performed, using the value to be added
- After the search is completed, the location for the new value is known

Inserting into a B+ Tree (cont.)



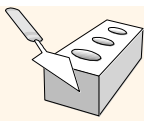
- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Inserting into a B+ Tree (cont.)



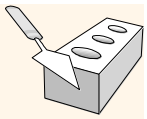
- If the tree is empty, add to the root
- Once the root is full, split the data into 2 leaves, using the root to hold keys and pointers

Inserting into a B+ Tree (cont.)



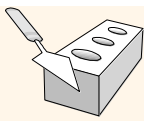
- Example:
  - Suppose we had a B+ tree with n = 3
    - 2 keys max. at each internal node
    - 3 pointers max. at each internal node
    - Internal nodes include the root

Inserting Into B+ Trees (cont.)

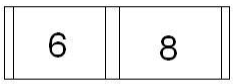


- Case 1: Empty root
- Insert 6

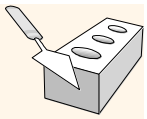
Inserting into B+ Trees (cont.)



- Case 2: Full root
- Suppose we have this root:

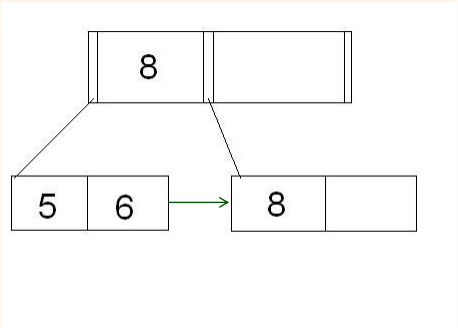


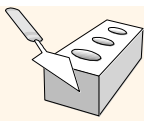
- In order to insert another number, like 5, we must split the root and create a new level.



Inserting into B+ Trees (cont.)

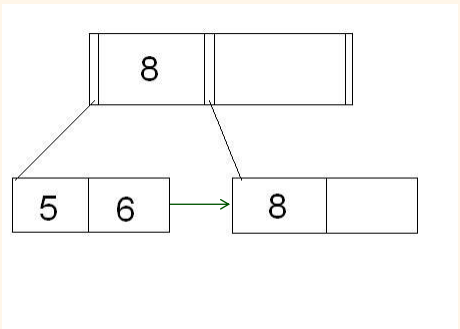
- After splitting the root, we would end up with this:



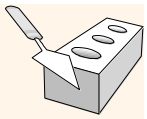


Inserting into B+ Trees (cont.)

- Case 3: Adding to a full node
- Suppose we wanted to insert 7 into our tree:

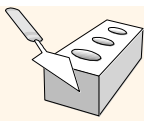


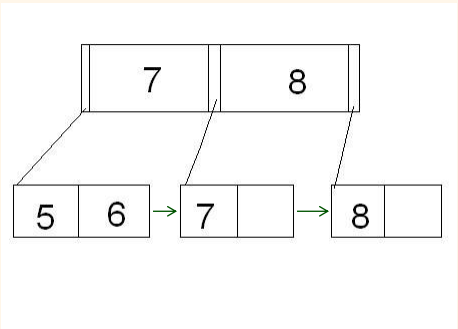
Inserting into B+ Trees (cont.)



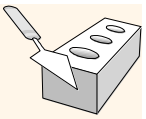
- 7 goes with 5 and 6. However, since each node can only hold a maximum of 2 keys, we can take the median of all 3 (which would be 6), keep it with the left (5), and create a new leaf for 7.
- An alternative way is to keep the median with the right and create a new leaf for 5.

Inserting into B+ Trees (cont.)

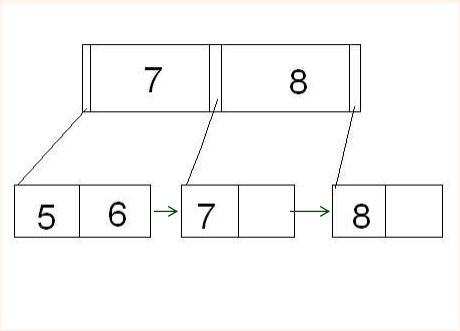




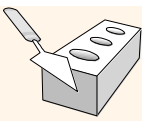
Inserting into B+ Trees (cont.)



- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
- Using the last tree, suppose we were to insert 4.

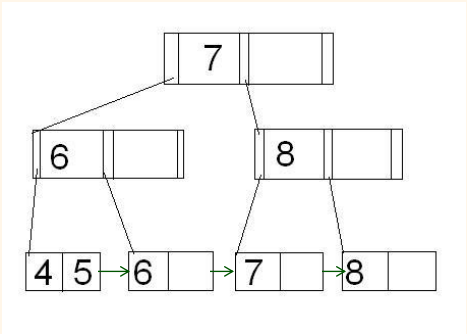


Inserting into B+ Trees (cont.)



- We would need to split the leftmost leaf, which would require another split of the root.
- A new root is created with the pointers referencing the old split root.

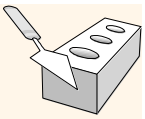
Inserting into B+ Trees (cont.)



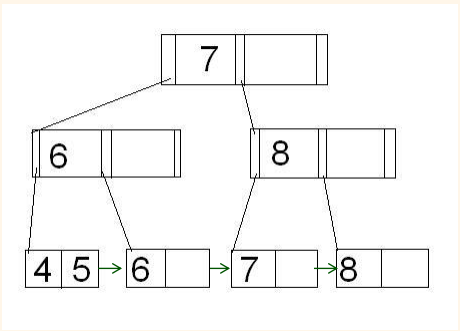
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* (*adjacent node with same parent as L*).
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

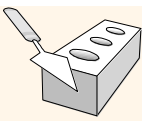
Deleting from B+ Trees (cont.)



- Take the previous example:



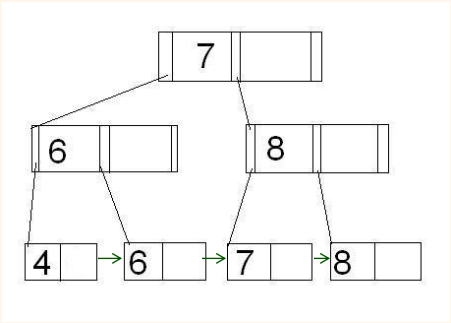
Deleting from B+ Trees (cont.)



- Suppose we want to delete 5.

- This would not require any rebalancing since the leaf that 5 was in still has 1 element in it.

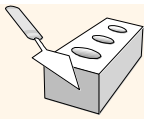
Deleting from B+ Trees (cont.)



Deleting from B+ Trees (cont.)

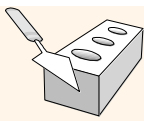
- Suppose we want to remove 6.
- This would require rebalancing, since removing the element 6 would require removal of the entire leaf (since 6 is the only element in that leaf).

Deleting from B+ Trees (cont.)

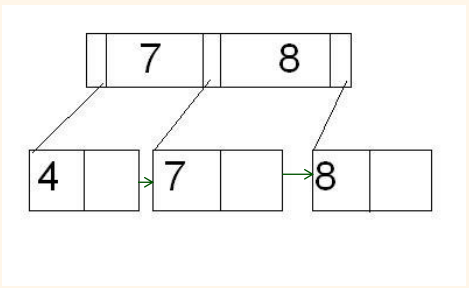


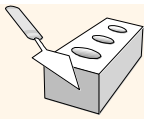
- Once we remove the leaf node, the parent of that leaf node no longer follows the rule.
- It has only 1 child, which is less than the 2 required (ceil(3/2) = 2).
- Then, the tree must be compacted in order to enforce this rule.

Deleting from B+ Trees (cont.)



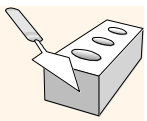
- The end product would look something like this:





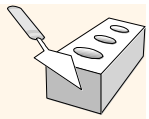
Hash-Based Indexes

Chapter 11



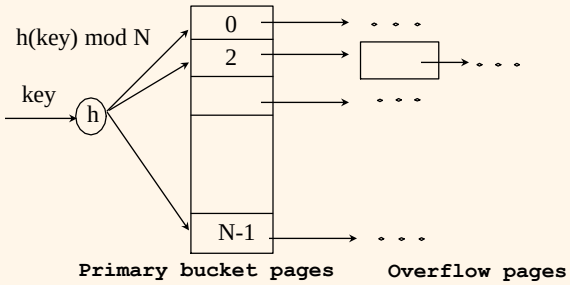
Introduction

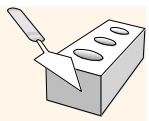
- *As for any index, 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - **<k**, rid of data record with search key value **k>**
  - **<k**, list of rids of data records with search key **k>**
  - Choice orthogonal to the *indexing technique*
- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.



Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)

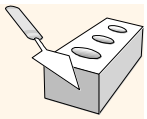




Static Hashing (Contd.)

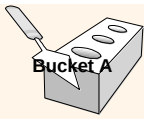
- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

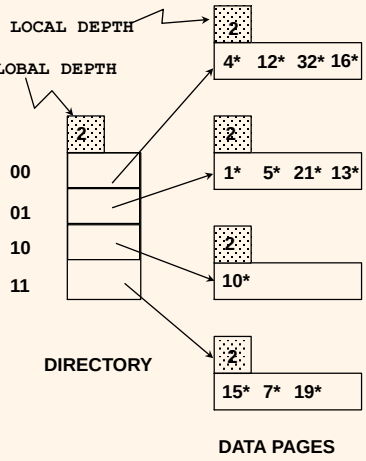
Extendible Hashing



- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

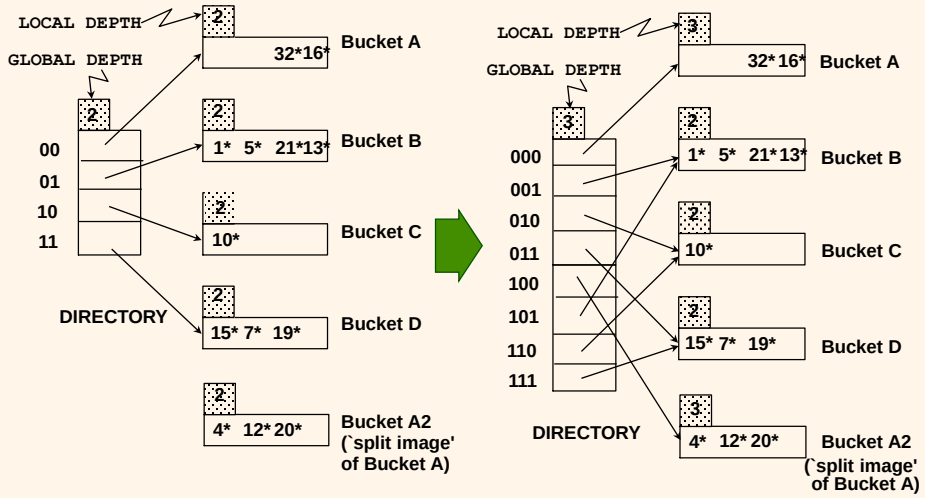
Example





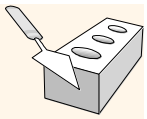
- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
- **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

*Insert **h**(r)=20 (Causes Doubling)*

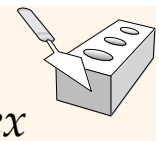


Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **3** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)



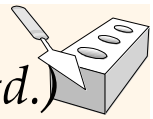
Discussion on Indices



Alternatives for Data Entry k in Index*

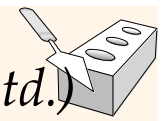
- In a data entry k* we can store:
  - Data record with key value **k,** or
  - **<k**, rid of data record with search key value **k>**, or
  - **<k**, list of rids of data records with search key **k>**
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

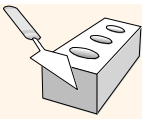


- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

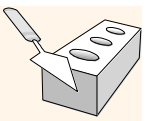


- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.



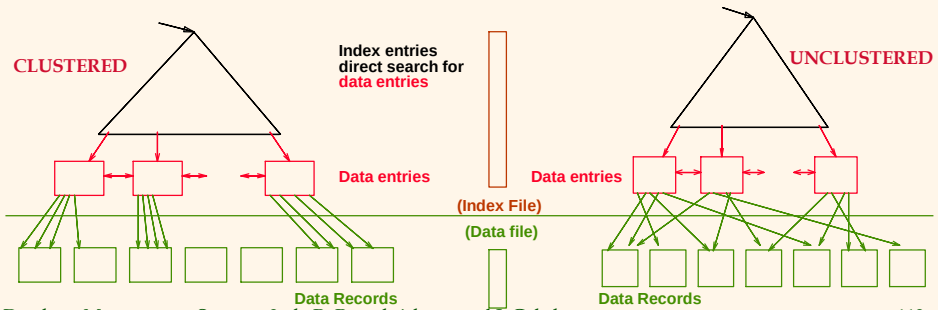
Index Classification

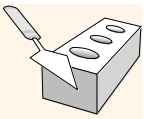
- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!



Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



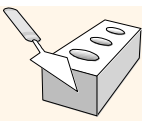


Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

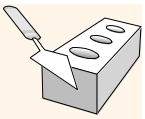
- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

** Good enough to show the overall trends!*



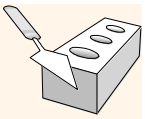
External Sorting

Chapter 13



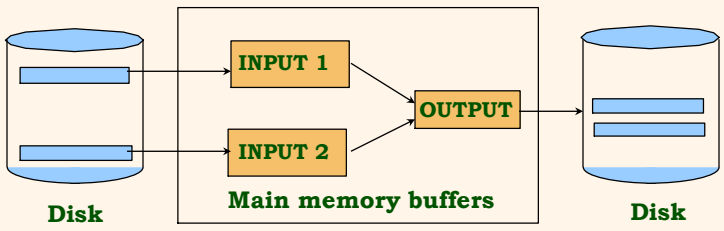
Why Sort?

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.
- Problem: sort 1Gb of data with 1Mb of RAM.
  - why not virtual memory?

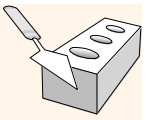


2-Way Sort: Requires 3 Buffers

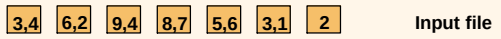
- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



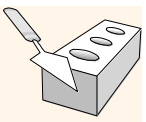
Two-Way External Merge Sort



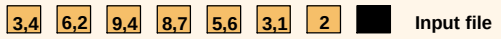
- Each pass we read + write each page in file.



Two-Way External Merge Sort

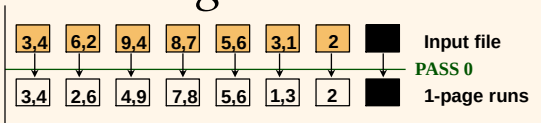


- Each pass we read + write each page in file.



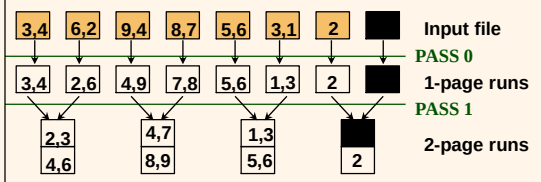
Two-Way External Merge Sort

- Each pass we read + write each page in file.



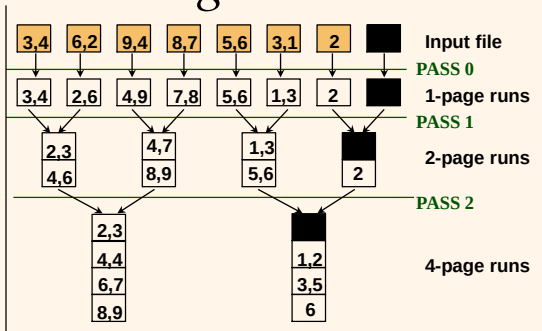
Two-Way External Merge Sort

- Each pass we read + write each page in file.



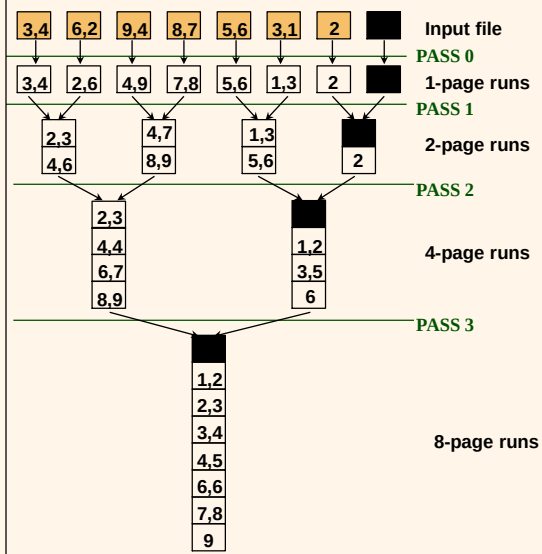
Two-Way External Merge Sort

- Each pass we read + write each page in file.



Two-Way External Merge Sort

- Each pass we read + write each page in file.

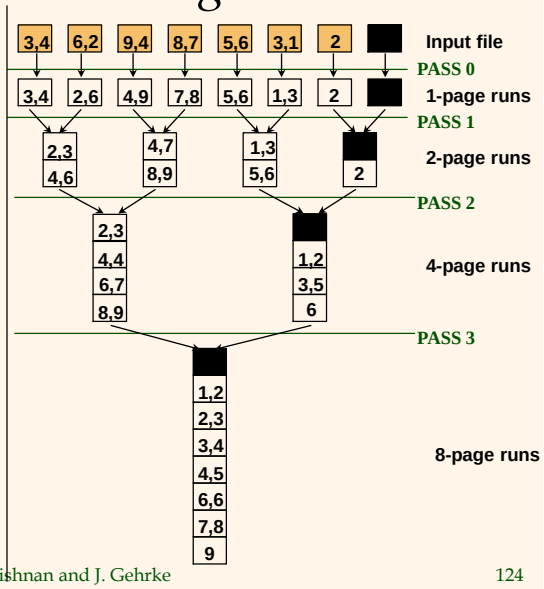


Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  $= \lceil \log_2 N \rceil + 1$
- So total cost is:

$$2N(\lceil \log_2 N \rceil + 1)$$

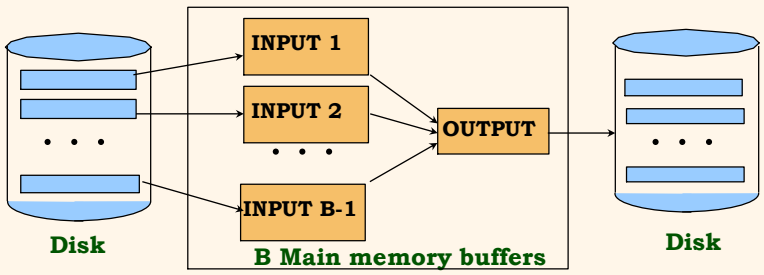
- *Idea:* ***Divide and conquer:*** sort subfiles and merge

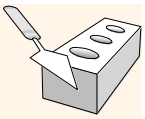


General External Merge Sort

**** More than 3 buffer pages. How can we utilize them?***

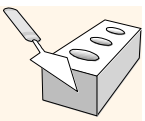
- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
  - Pass 2, …, etc.: merge B-1 runs.





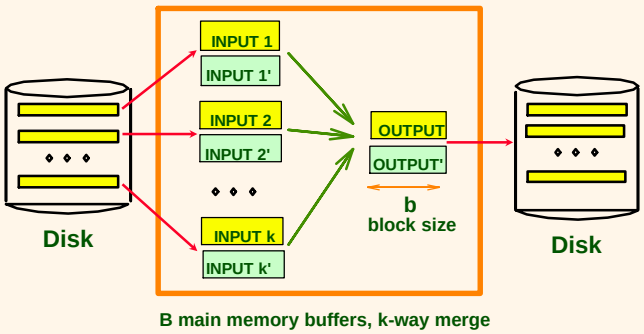
Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{B-1}\lceil N / B\rceil\rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5\rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4\rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

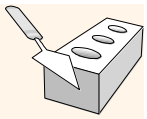


Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.

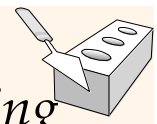


B main memory buffers, k-way merge



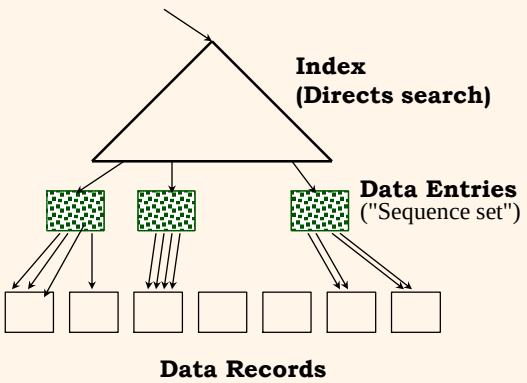
Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- ***Is this a good idea?***
- Cases to consider:
  - B+ tree is clustered — ***Good idea!***
  - B+ tree is not clustered — ***Could be a very bad idea!***



Clustered B+ Tree Used for Sorting

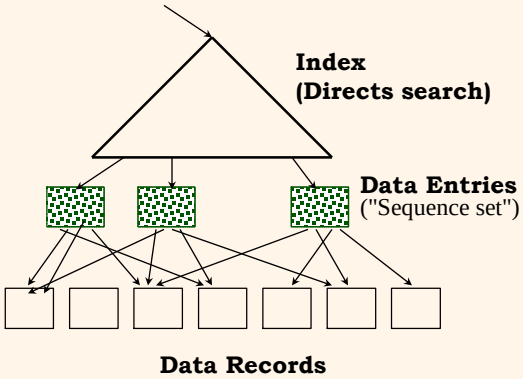
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

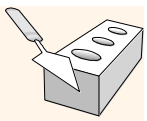


* ***Always better than external sorting!***

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!

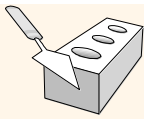




Evaluation of Relational Operations

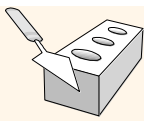
Chapter 14, Part A (Joins)

Relational Operations



- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

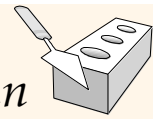
Schema for Examples



Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

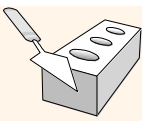
Equality Joins With One Join Column



```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

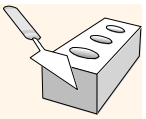
- In algebra: R ⋈ S. Common! Must be carefully optimized. R × S is large; so, R × S followed by a selection is inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

Simple Nested Loops Join



```
foreach tuple r in R do
        foreach tuple s in S do
                if ri == sj then add <r, s> to result
```

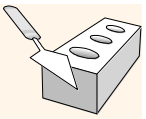
- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os.
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M*N = 1000 + 1000*500
  - If smaller relation (S) is outer, cost = 500 + 500*1000



Index Nested Loops Join

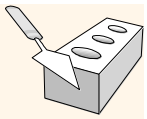
foreach tuple r in R do
  foreach tuple s in S where r_i == s_j do
    add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical)
  - Unclustered index: up to 1 I/O per matching S tuple.



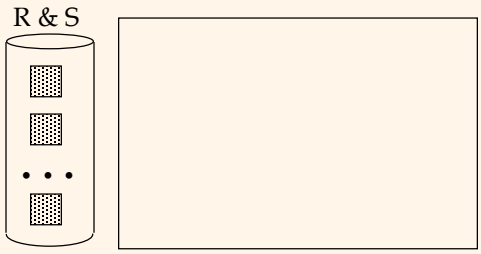
Examples of Index Nested Loops

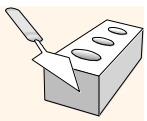
- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.



Block Nested Loops Join

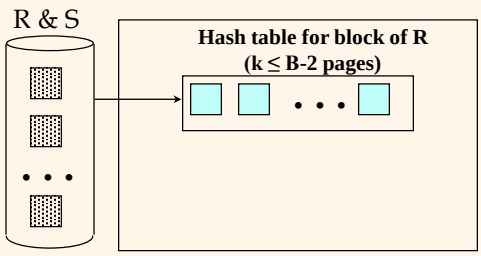
- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.

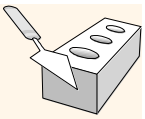




Block Nested Loops Join

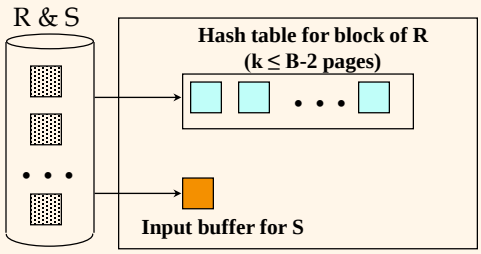
- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.

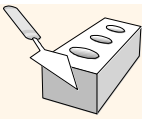




Block Nested Loops Join

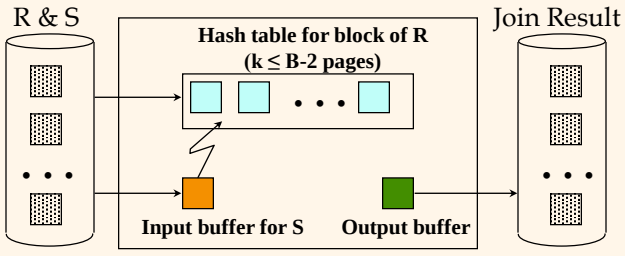
- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.



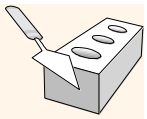


Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



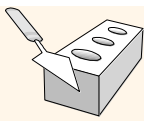
Block Nested Loops Join



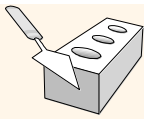
```
foreach block of B − 2 pages of R do
        foreach page of S do {
        for all matching in-memory tuples r in R-block and s in S-page,
        add <r, s> to result
}
```

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \#\ of\ pages\ of\ outer\ /\ blocksize \rceil$
  - total cost: M +N*$\lceil M/(B-2) \rceil$ I/Os

Examples of Block Nested Loops

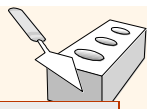


- With Reserves as outer, and space for 100 pages of R:
  - Cost of scanning R: 1000 I/Os; we need 1000/100= 10 blocks
  - Per block of R, we scan Sailors (S); (1000/100)*500 I/Os.
  - Total cost is 1000 + (1000/100)*500 I/Os
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S: 500 I/Os; a total of 500/100 = 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.



Sort-Merge Join ($R \underset{i=j}{\bowtie} S$)

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)



Example of Sort-Merge Join

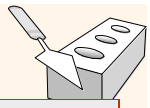
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname

Example of Sort-Merge Join



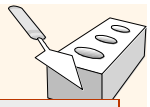
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname

Example of Sort-Merge Join



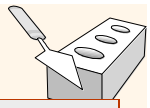
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname

Example of Sort-Merge Join



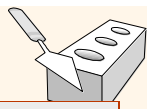
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname

Example of Sort-Merge Join



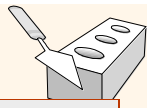
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy

Example of Sort-Merge Join



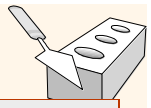
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy

Example of Sort-Merge Join



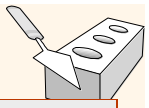
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy

Example of Sort-Merge Join



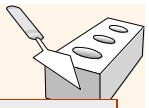
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy

Example of Sort-Merge Join



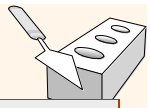
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin

Example of Sort-Merge Join



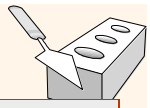
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
31	lubber	8	55.5	102	10/12/96	lubber

Example of Sort-Merge Join



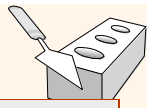
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
…	…	…	…	…	…	…

Example of Sort-Merge Join



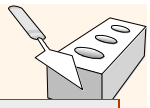
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
…	…	…	…	…	…	…

Example of Sort-Merge Join



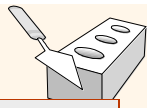
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
…	…	…	…	…	…	…

Example of Sort-Merge Join



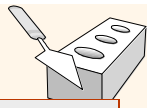
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
…	…	…	…	…	…	…

Example of Sort-Merge Join



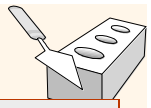
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

output

sid	sname	rating	age	bid	day	rname
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy
31	lubber	8	55.5	101	10/10/96	dustin
…	…	…	…	…	…	…
58	rusty	10	35.0	103	11/12/96	dustin

Example of Sort-Merge Join



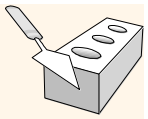
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

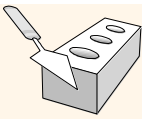
- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(*BNL cost: 2500 to 15000 I/Os*)

Refinement of Sort-Merge Join

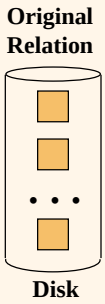


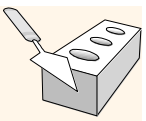
- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With B > $\sqrt{L}$, where L is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.



Hash-Join

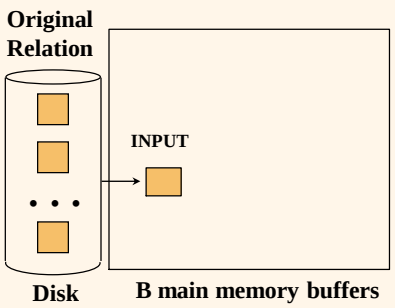
- Partition both relations using hash fn **h**





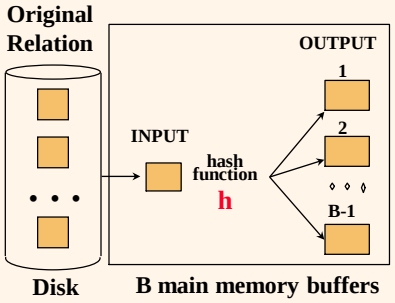
Hash-Join

- Partition both relations using hash fn **h**



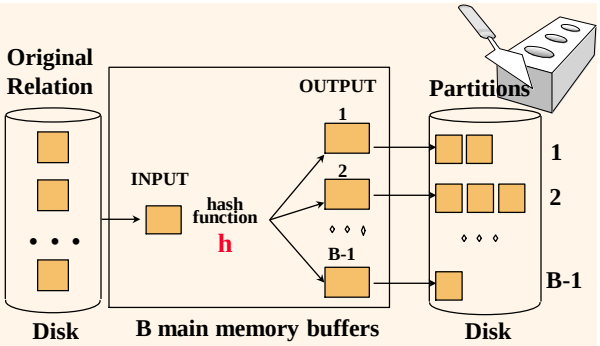
Hash-Join

- Partition both relations using hash fn **h**



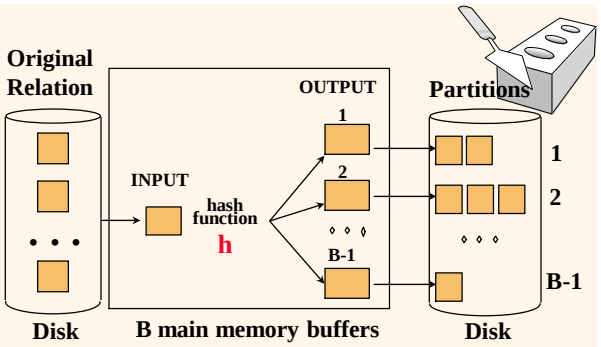
Hash-Join

- Partition both relations using hash fn **h**



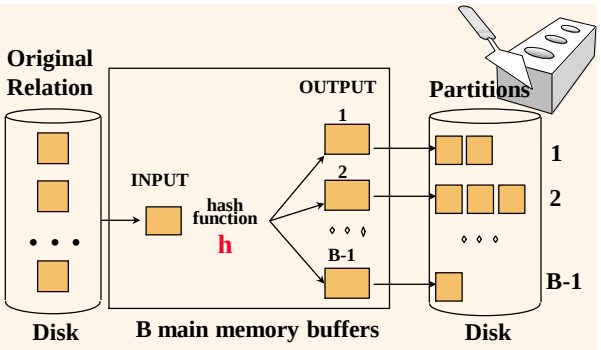
Hash-Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.

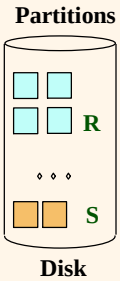


Hash-Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.

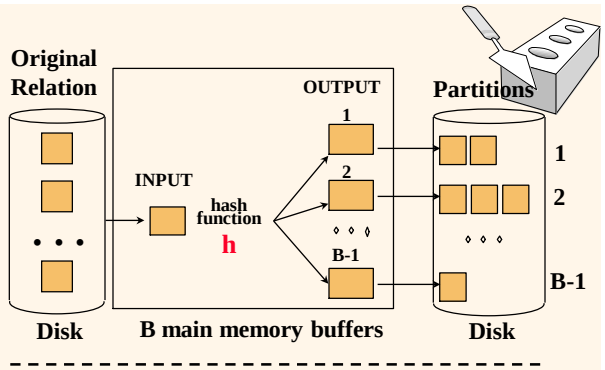


- Read in one of the k partitions of R, hash it using **h2 (<> h!)**.

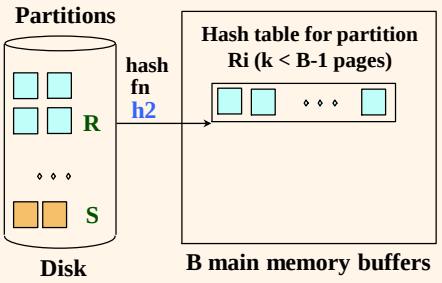


Hash-Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.

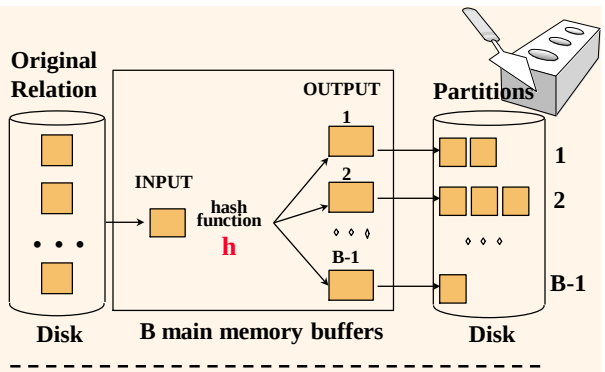


- Read in one of the k partitions of R, hash it using **h2 (<> h!)**.

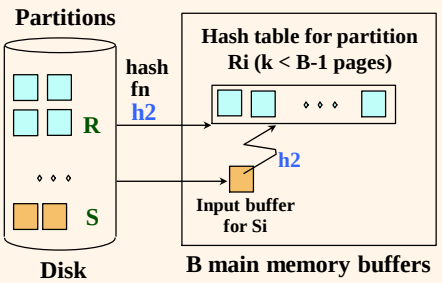


Hash-Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.

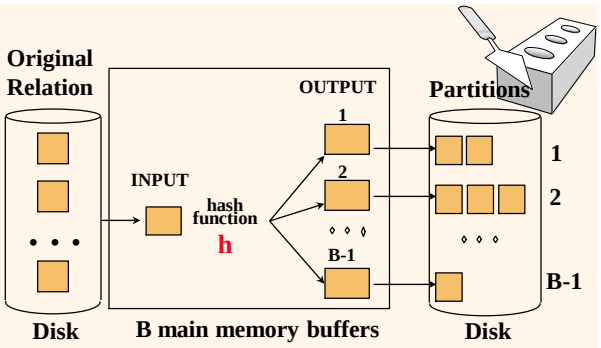


- Read in one of the k partitions of R, hash it using **h2 (<> h!)**.
- Scan matching partition of S, search for matches.

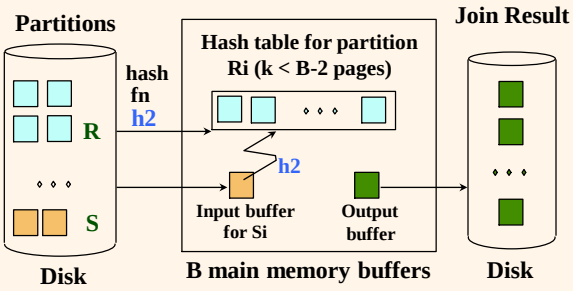


Hash-Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.



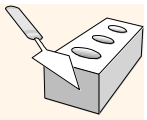
- Read in one of the k partitions of R, hash it using **h2 (<> h!)**.
- Scan matching partition of S, search for matches.



Observations on Hash-Join

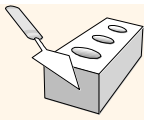
- #partitions k < B-1 (why?), and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) < B-2, i.e., we need B > $\sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join



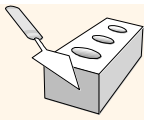
- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- total cost: 3(M+N) I/Os

Cost of Hash-Join



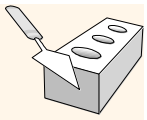
- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- total cost: 3(M+N) I/Os
- In our running example, this is a total of 4500 I/Os.
  - completes in less than 60sec (assume 10ms per disk I/O)

Cost of Hash-Join



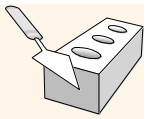
- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- total cost: 3(M+N) I/Os
- In our running example, this is a total of 4500 I/Os.
  - completes in less than 60sec (assume 10ms per disk I/O)
- Compare with simple nested loops join: 140 *hours* (1000 + 100*1000*500 I/Os)

Cost of Hash-Join



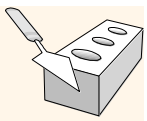
- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- total cost: 3(M+N) I/Os
- In our running example, this is a total of 4500 I/Os.
  - completes in less than 60sec (assume 10ms per disk I/O)
- Compare with simple nested loops join: 140 *hours* (1000 + 100*1000*500 I/Os)
- this difference underscores importance of using a good join algorithm!

Cost of Hash-Join

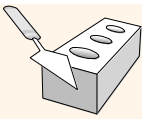


- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have cost 3(M+N) I/Os.
  - Hash Join superior on this count if relation sizes differ greatly.
  - Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew.
  - result of Sort-Merge is sorted.

General Join Conditions

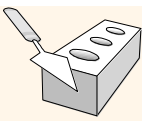


- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

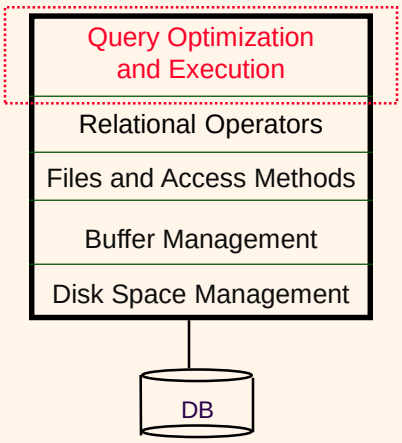


Relational Query Optimization

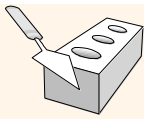
Chapter 15



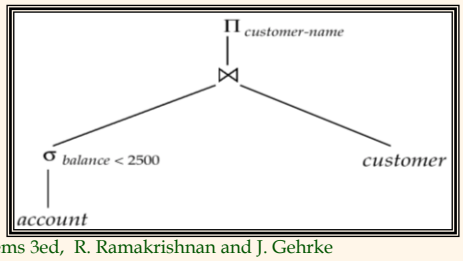
Context



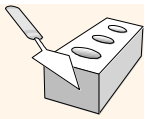
Query Plan



- a tree representation of a relational algebra query expression
  - implicitly encodes the order of operator evaluation
- this example is a logical query plan

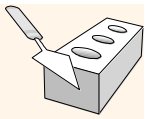


Overview of Query Optimization



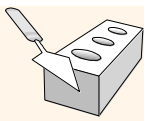
- *Plan: Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan. Practically: Avoid worst plans!
- We will study the System R approach.

Evaluation of Expressions



- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression (operator) 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

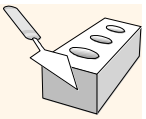
Schema for Examples



Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Query Blocks: Units of Optimization



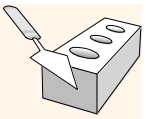
- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an over-simplification, but serves for now.)

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2
     GROUP BY  S2.rating)
```

Outer block *Nested block*

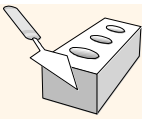
- For each block, the plans considered are:
  - All available access methods, for each reln in FROM clause.
  - All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)

Query Optimization



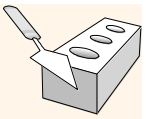
- Query Rewriting:
  - Given a relational algebra expression produce and equivalent expression that can be evaluated more efficiently
- Plan generator:
  - Choose the best algorithm for each operator given statistics about the database, main memory constraints and available indices

Transformation of Relational Expressions



- Two RA expressions are equivalent if they produce the same results on the same inputs
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples
- An equivalence rule says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules



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

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

2. Selection operations are commutative.

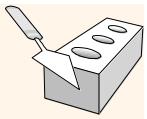
$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

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

$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(R))\ldots)) = \Pi_{t_1}(R)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(R_1 \text{ X } R_2) = R_1 \bowtie_{\theta} R_2$
   b. $\sigma_{\theta 1}(R_1 \bowtie_{\theta 2} R_2) = R_1 \bowtie_{\theta 1 \wedge \theta 2} R_2$

Equivalence Rules (Cont.)



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

$$R_1 \bowtie_\theta R_2 = R_2 \bowtie_\theta R_1$$

6. (a) Natural join operations are associative:

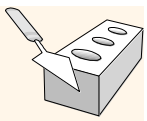
$$(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$$

(b) Theta joins are associative in the following manner:

$$(R_1 \bowtie_{\theta 1} R_2) \bowtie_{\theta 2 \wedge \theta 3} R_3 = R_1 \bowtie_{\theta 1 \wedge \theta 3} (R_2 \bowtie_{\theta 2} R_3)$$

where θ_2 involves attributes from only R_2 and R_3.

Equivalence Rules (Cont.)



7. The selection operation distributes over the theta join operation under the following two conditions:

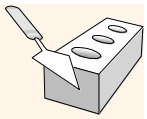
(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (R_1) being joined.

$$\sigma_{\theta 0}(R_1 \bowtie_\theta R_2) = (\sigma_{\theta 0}(R_1)) \bowtie_\theta R_2$$

(b) When θ_1 involves only the attributes of R_1 and θ_2 involves only the attributes of R_2.

$$\sigma_{\theta 1 \wedge \theta 2}(R_1 \bowtie_\theta R_2) = (\sigma_{\theta 1}(R_1)) \bowtie_\theta (\sigma_{\theta 2}(R_2))$$

Equivalence Rules (Cont.)



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

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

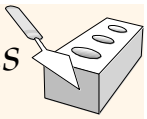
$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

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

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

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

Example with Multiple Transformations



- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

$\Pi_{cname}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$

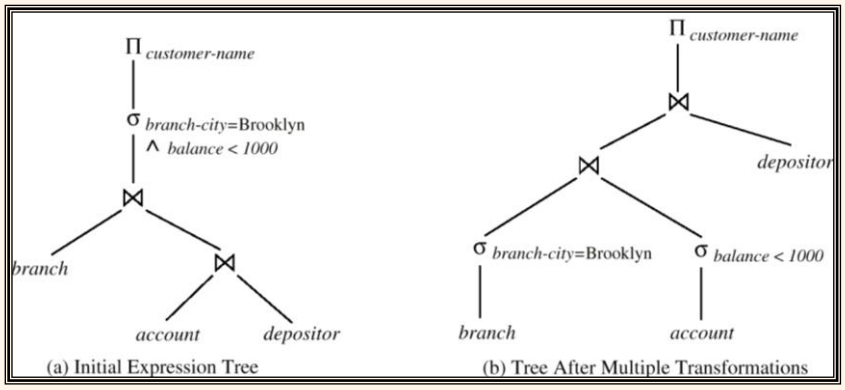
- Transformation using join associatively (Rule 6a):

$\Pi_{cname}((\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$

Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

$\Pi_{cname}((\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch)) \bowtie \sigma_{balance > 1000}(account)) \bowtie depositor)$

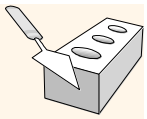
Multiple Transformations (Cont.)



Enumeration of Alternative Plans

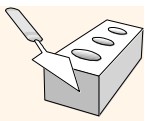
- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

Cost Estimation

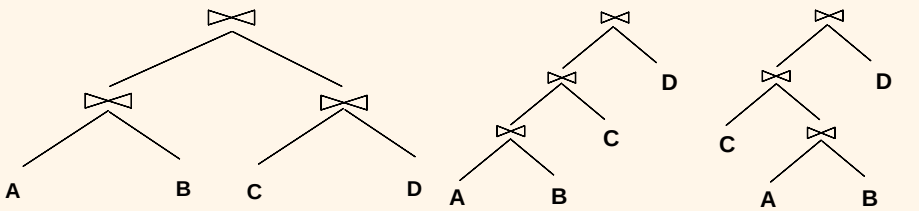


- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

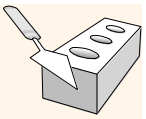
Queries Over Multiple Relations



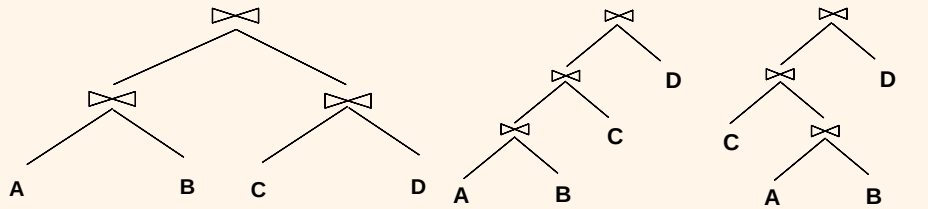
- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - are all left-deep trees fully pipelined?



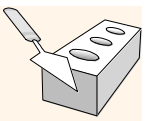
Queries Over Multiple Relations



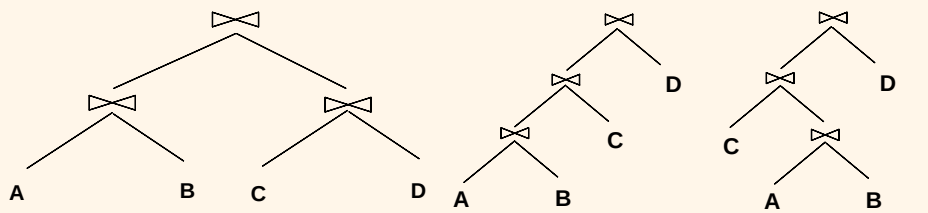
- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined



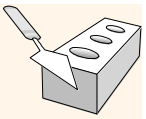
Queries Over Multiple Relations



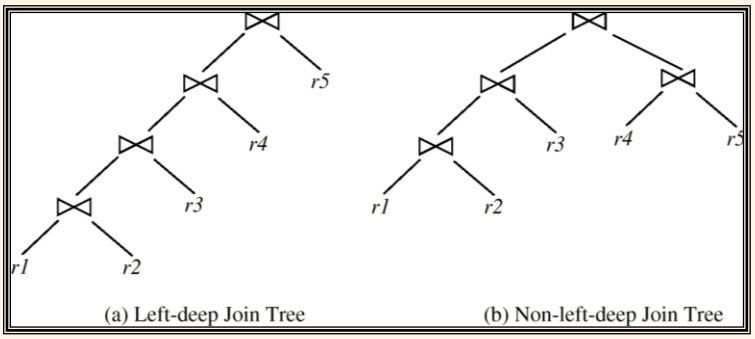
- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



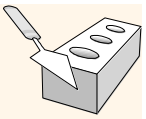
Left Deep Join Trees



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



Join Ordering



- For all relations r_1, r_2, and r_3,

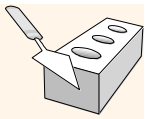
  $$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

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

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

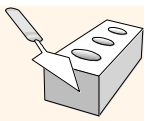
  so that we compute and store a smaller temporary relation.

Cost-Based Optimization



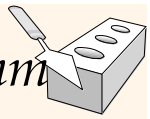
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $[2(n-1)]!/(n-1)!$ different join orders for the expression above. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic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

Dynamic Programming in Optimization



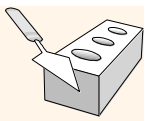
- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations
    - consider all possible plans of form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
    - recursively compute costs for joining subsets of S to find cost of each plan
    - choose the cheapest of the $2^n - 1$ alternatives.

Join Order Optimization Algorithm



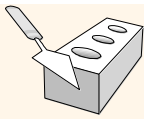
```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    for each non-empty subset S1 of S such that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```


Cost of Optimization



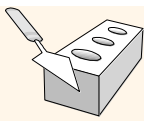
- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
- Space complexity is $O(2^n)$
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Heuristic Optimization



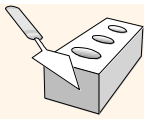
- Cost-based optimization is expensive, even with dynamic programming.
- Systems use *heuristics* to reduce number of choices that must be made in a cost-based fashion.

Heuristic Optimization



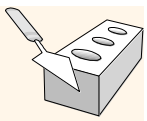
- Heuristic optimization transforms the query-tree using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Enumeration of Left-Deep Plans



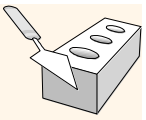
- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.

Enumeration of Plans (Contd.)



- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator.
- An N-1 way plan is not combined with an additional relation if there is no join condition between them, unless all predicates in WHERE have been used up.
  - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables.

Highlights of System R Optimizer



- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.