Implementation of DBMS

Justus Klingemann

Outline

This course explores the internals of database management systems

- Address the architectures and implementation issues relevant for these systems

Complementary Topics

- Database design:
  - The informal, high-level specification of the schema of a database
  - Notations like the Entity-Relationship-Model
  - The implementation of designs in the data-definition portion of SQL
- Database programming
  - Writing Queries and database modification commands using appropriate languages, especially SQL

Why this Lecture?

Databases form the backbone of every modern information system

- A robust database management system (DBMS) is crucial for these systems.
- The knowledge of the internals of a DBMS forms the prerequisite for building and extending a DBMS as well as for building the DBMS part of a larger application in a robust fashion.
- In addition, it helps to understand the role of the different parameters of commercial DBMS and thus, tune these parameters in a way that results in a robust and performant system.

Literature

Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database System Implementation
OR
Database Systems: The Complete Book

Saake, G.: Heuer, A., K.-U. Sattler: Datenbanken: Implementierungstechniken

Theo Härder, Erhard Rahm: Datenbanksysteme - Konzepte und Techniken der Implementierung

Shasha, D., Bonnet, P.: Database Tuning: Principles, Experiments and Troubleshooting Techniques

What should a DBMS achieve?

Codd‘s Nine Rules

- Integration: uniform, non-redundant data management
- Operations: store, search, modify
- Catalogue: access database descriptions in a data dictionary
- Views for different users
- Ensuring Integrity: Correctness of the content of the database
- Data security: Rule out unauthorized access
- Transactions: Bundle several database operations to one unit
- Synchronization: coordinate parallel transactions
- Availability of data: Recover data after system failures

How NOT to implement a DBMS

Megatron 3000: An example provided by Hector Garcia Molina

Megatron 3000 Implementation Details

Relations stored in files (ASCII)

e.g., relation R is in /usr/db/R

```
Smith # 123 # CS
Jones # 522 # EE
  ⋮
```


Megatron 3000 Implementation Details

Directory file (ASCII) in /usr/db/directory

```
R1 # A # INT # B # STR …
R2 # C # STR # A # INT …
  ⋮
```

Megatron 3000 Sample Sessions

```
% MEGATRON3000
   Welcome to MEGATRON 3000!
&
  :
  :
& quit
%
```


Megatron 3000 Sample Sessions

```
& select *
  from R #

    Relation R
  A       B     C
  SMITH   123   CS

&
```


Megatron 3000 Sample Sessions

```
& select A,B
  from R,S
  where R.A = S.A and S.C > 100 #

   A     B
  123   CAR
  522   CAT

&
```


Megatron 3000

To execute "`select * from R where condition`":

Megatron 3000

To execute "**`select * from R where condition`**":

(1) Read dictionary to get R attributes

(2) Read R file, for each line:

(a) Check condition

(b) If OK, display

Megatron 3000

To execute "**`select A,B from R,S where condition`**":

Megatron 3000

To execute "**`select A,B from R,S where condition`**":

(1) Read dictionary to get R,S attributes

(2) Read R file, for each line:

(a) Read S file, for each line:

(i) Create join tuple

(ii) Check condition

(iii) Display if OK

What's wrong with the Megatron 3000 DBMS? (1)

Tuple layout on disk

e.g., - Change string from 'Cat' to 'Cats' and we have to rewrite file
- Deletions are also expensive

Search expensive; no indexes

e.g., - Cannot find tuple with given key quickly
- Always have to read full relation

Brute force query processing

e.g.,
```
select *
from R,S
where R.A = S.A and S.B > 1000
```
- Do select first?
- More efficient join?

No buffer manager

e.g., Need caching

What's wrong with the Megatron 3000 DBMS? (2)

No concurrency control

No reliability

e.g., - Can lose data
- Can leave operations half done

No security

e.g., - File system insecure
- File system security is coarse

No application program interface (API)

e.g., How can a payroll program get at the data?

No GUI

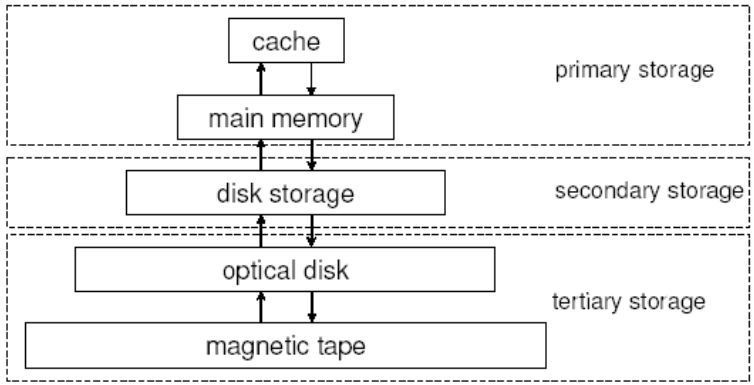
Cannot interact with other DBMSs.

Poor dictionary facilities

Storage Hierarchy

Storage Hierarchy

Primary, secondary and tertiary storage



Primary Storage

Primary Storage consists of main memory and processor cache

Very fast, e.g. for DDR4-3200 memory:

- sequential read: 50 GByte/s
- latency: 100ns

Data can be directly processed by CPU

Access to data with fine granularity: each byte can be addressed

Number of accessible bytes depends on address scheme: 32-bit address scheme implies that only 2^{32} bytes are addressable

Volatile, non-reliable storage media

Secondary Storage

hard disk storage or SSD

stable, non-volatile, reliable

much larger, e.g. 10 TByte per medium

By orders of magnitude cheaper

Data can not be directly processed

Access granularity is coarse: blocks of e.g., 4 KBytes

Much slower, even for modern drives:

- for HDD: sequential read: up to 280 MByte/s, latency: 1-30 ms
- for SSD: sequential read: up to 7000 MByte/s, latency: 10-100 µs

Necessary:

- good buffer management (high hit ratio)
- good query management

SDD versus HDD

Advantages SSD:

- faster
- physically smaller

Advantages HDD:

- larger capacity per drive
- cheaper

In particular the cost limits the use of SSDs to store large amounts of data

- SSDs dominate for storing boot-partitions and small amounts of data
- HDDs still frequently for storing large databases

Sales figures for 2020:

- the number of sold drives is quite similar for HDDs and SSDs
- the sold capacity of HDDs is around five times the capacity of SSDs

Tertiary Storage

Usage:

- for long-term archival storage or
- short-term logging (journals) of database updates

Secondary storage is too expansive for this purpose

Media: optical discs, magnetic tapes

Size of a magnetic tape: several terabytes

Medium is typically switched: “offline storage”

Disadvantage: access gap extremely large: manually access medium, insert medium

Performance Metrics (1)

Crucial for the performance of a DBMS is the transition between main memory (primary storage) and secondary storage

- to work with data, it has to be in primary storage (in a region called buffer), but:
- main memory is not persistent
- the main memory buffer is typically much smaller than the database. Consequence: We have to discard data from the buffer to load other parts of the database into the buffer; modified data has to transferred to secondary storage before it can be discarded
- Result: Data has to be transferred frequently between main memory and secondary storage.

Data is transferred between main memory and secondary storage in units called blocks (size depends on media, e.g., 4KByte).

Performance Metrics (2)

A disk I/O (read or write of a block) is very expensive compared with what is likely to be done with the block once it arrives in main memory.

- Perhaps 1,000,000 machine instructions in the time to do one random disk I/O.
- Random block accesses is the norm if there are several processes accessing disks, and the disk controller does not schedule accesses carefully.

Reasonable model of computation that requires secondary storage: count only the disk I/O's.

Good DBMS algorithms

a) Try to make sure that if we read a block, we use much of the data on the block.

b) Try to put blocks that are accessed together in physically adjacent locations.

c) Try to buffer commonly used blocks in main memory.

Sorting Example

Setup:

- 10^7 records of 100 bytes = 10^9 bytes file.
  - Stored on a disk with 4KByte blocks, each holding 40 records + header information.
  - Entire file takes 250,000 blocks. 10^9 /4000 = 250,000 Blocks
- 50MByte available main memory = 12,800 blocks ≈ 1/20th of file.
- Task: Sort records of file by primary key field.

50 MB= 50*1024*1024 / 4096 =12,800 blocks buffers

Merge Sort

Common main memory sorting algorithms do not perform well when you take disk I/O's into account. Variants of Merge Sort do better.

- Merge = take two sorted lists and repeatedly chose the smaller of the 'heads" of the lists (head = first of the unchosen).
- Example: merge 1,3,4,8 with 2,5,7,9 = 1,2,3,4,5,7,8,9.
- Merge Sort based on recursive algorithm:
  - divide records into two parts;
  - recursively merge sort the parts, and merge the resulting lists.

Two Phase, Multiway Merge Sort (1)

Plain merge sort is still not very good in number of disk I/O's.

- $\log_2(n)$ passes, so each record is read/written from disk $\log_2(n)$ times.

Better variant: 2PMMS

- 2 reads + 2 writes per block.

Two Phase, Multiway Merge Sort (2)

Phase 1

- 1. Fill main memory with records.
- 2. Sort using favorite main memory sort.
- 3. Write sorted sublist to disk.
- 4. Repeat until all records have been put into one of the sorted lists.

Two Phase, Multiway Merge Sort (3)

Phase 2

- Use one buffer for each of the sorted sublists and one buffer for an output block.
- Initially load input buffers with the first blocks of their respective sorted lists.
- Repeatedly run a competition among the first unchosen records of each of the buffered blocks.
  - Move the record with the least key to the output block; it is now "chosen."
- Manage the buffers as needed:
  - If an input block is exhausted, get the next block from the same file.
  - If the output block is full, write it to disk.

Analysis

We count the I/O's:

- File stored on 250,000 blocks, read and written once in each phase.
- 4 * 250,000 I/O's = 1,000,000 I/O's

Depending on how the data is organized on the storage media, the cost of an individual I/O can vary!!!

Remaining question: How many records can you sort with 2PMMS, under our assumptions about records, main memory, and the disk? What would you do if there were more?

Extension of 2PMMS to Larger Relations

Possible size of relation can be calculated as follows

- block size: B bytes
- main memory available for buffering blocks: M bytes
- size of record: R bytes

Then

- number of available buffer blocks in main memory: M/B
- during the second phase all but one buffer may be devoted to one of the sorted sublists; the remaining buffer is for the output block
  - maximum number of sorted sublists is (M/B) – 1
- Each sublist completely fits into main memory in phase one
  - sublist can contain M/R records
- Total number of records we can sort: $(M/R)\ ((M/B) - 1) \approx M^2 / (R * B)$
- In our example: $6.71 * 10^9$ records

If this is not enough: add a third pass

Representing Data Elements

Physical Layout of Data

In relational terms:

- Field = sequence of bytes representing the value of an attribute in a tuple.
- Record = sequence of bytes divided into fields, representing a tuple.
- File = collection of blocks used to hold a relation = set of tuples or records, respectively.

In object-oriented terms:

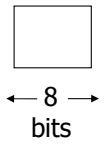
- Field represents an attribute or relationship.
- Record represents an object.
- File represents extent of a class.

The Mapping Problem

What are the data items we want to store?

- a salary
- a name
- a date
- a picture

⇨ What we have available: Bytes



Numbers

Integer: 2/4 bytes

e.g., 35 is

00000000 00100011

- Real, floating point
  n bits for mantissa, *m* for exponent….

Characters

→ various coding schemes suggested,
most popular is ascii

Example:
A: 1000001
a: 1100001
5: 0110101
LF: 0001010

Boolean Values

Boolean

e.g., TRUE 1111 1111
FALSE 0000 0000

Application specific Enumerations
e.g., RED → 1 GREEN → 3
BLUE → 2 YELLOW → 4 …

⇨ Can we use less than 1 byte/code?

Yes, but only if desperate...

Date and Time

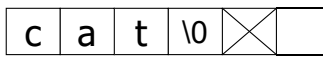
Dates

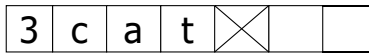
e.g.: - Integer, # days since Jan 1, 1900
- 8 characters, YYYYMMDD
(not YYMMDD!)
- 7 characters, YYYYDDD
- SQL: YYYY-MM-DD

Time

e.g. - Integer, seconds since midnight
- characters, HHMMSSFF

Strings of Characters

- Null terminated
e.g., 

- Length given
e.g., 

- Fixed length

Records

Consider fixed-length records first

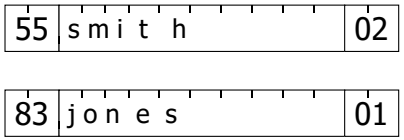
Record the consists of

- Space for each field of the record.
  - Sometimes, it is required to align fields starting at a multiple of 4 or 8.

Example: Employee record

(1) E#, 2 byte integer
(2) E_name, 10 char.
(3) Dept, 2 byte code

Schema



Records

Record Header

Usually the fields of the record are preceded by a header

Header = space for information about the record, e.g.,

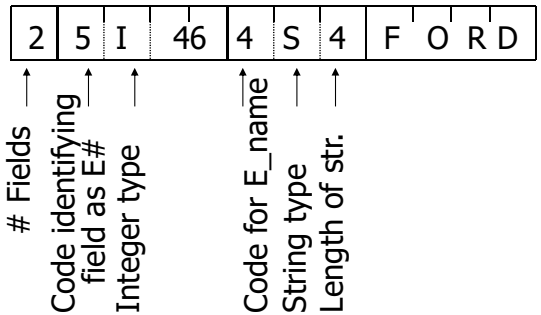
- record format (pointer to schema),
- record length,
- timestamp.

Variable-Length Records

Can occur in case of

- Fields that vary in length
- Repeating fields, e.g., a set of pointers represent a manymany relationship
- Variableformat records: field names are arbitrary
  - Important for selfdescribing data, information integration.

Example: variable format and length



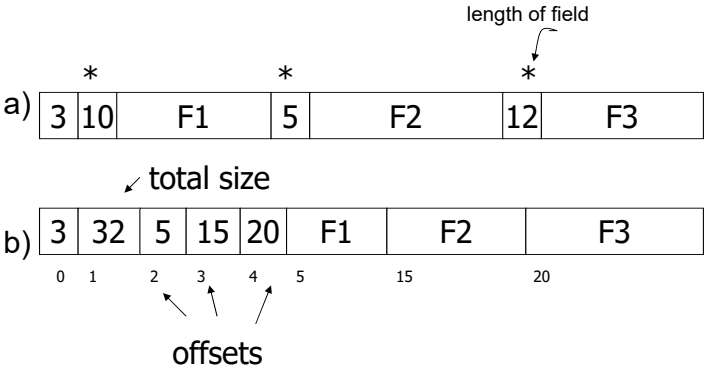
Field name codes could also be strings, i.e. TAGS

Example: Repeating Fields

Employee has one or more children

3	E_name: Fred	Child: Sally	Child: Tom

Internal Organization of Record



Strategy a) Each field is preceded by a number providing its length

Strategy b) The Record header contains a set of pointers to the variable length fields

Hybrid Format

Hybrid format

- one part is fixed, other variable

E.g.: All employees have E#, name, dept
other fields vary.

25	Smith	Toy	2	Hobby:chess	state:retired

# of var fields

Alternative realization

- Split Records Into Fixed/Variable Parts
- Fixed part has a pointer to space where current value of variable fields can be found.

Placing Records into Blocks

Structure of Blocks:

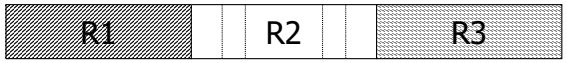
1. Block header = space for info such as:
   - Links to other blocks of a data structure.
   - Role info for this block, e.g., for which relation does the block hold tuples?
   - Directory of records in the block.
   - Block ID.
   - Timestamp.
2. Some number of records

Options for storing records in blocks

(1) separating records
(2) spanned vs. unspanned
(3) mixed record types – clustering
(4) split records
(5) sequencing
(6) addressing

Separating records

When does a record ends and the next starts?

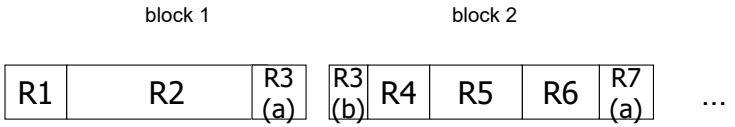
Block 

(a) no need to separate - fixed size recs.
(b) special marker
(c) give record lengths (or offsets)
- within each record
- in block header

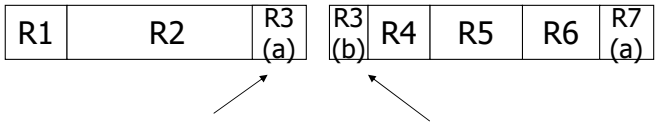
Spanned vs. Unspanned

Unspanned: records must be within one block

A spanned record can be divided between two blocks



Spanned records



need indication
of partial record
+ “pointer” to rest

need indication
of continuation

Spanned vs. Unspanned

Unspanned is much simpler, but may waste space…

Spanned essential if

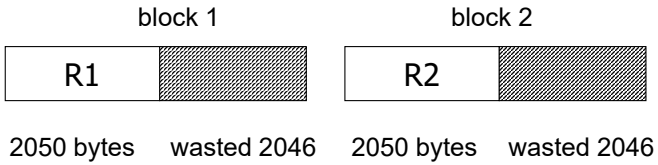
record size > block size

Example

10^6 records

each of size 2,050 bytes (fixed)

block size = 4096 bytes



And so on.....

Total wasted $\approx 2 \times 10^9$ Utilization $\approx 50\%$

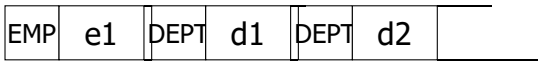
Total space $\approx 4 \times 10^9$

With 2,050-byte records and 4,096-byte blocks, only 1 record per block fits.

Mixed record types

Mixed: records of different types (e.g. EMPLOYEE, DEPT) allowed in same block

e.g., a block



Why do we want to mix?

Answer: CLUSTERING

Records that are frequently accessed together should be in the same block

Example

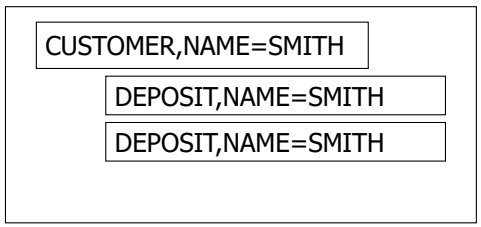
Q1: select A#, C_NAME, C_CITY, …

from DEPOSIT, CUSTOMER

where DEPOSIT.C_NAME =

CUSTOMER.C.NAME

a block



Options for storing records in blocks

If Q1 frequent, clustering is good

But if Q2 frequent

```
Q2:   SELECT *
        FROM CUSTOMER
```

CLUSTERING IS COUNTER PRODUCTIVE

"Clustering is counter productive":
This means that creating a clustered index (physically organizing the data on disk based on a certain column) will not improve performance for this query — and may even slow it down.

Split records

Typically for hybrid format

- Fixed part in one block
- Variable part in another block

Sequencing

Ordering records in file (and block) by some key value

⇒ sequential file

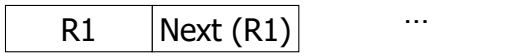
Why sequencing?

Typically to make it possible to efficiently read records in order

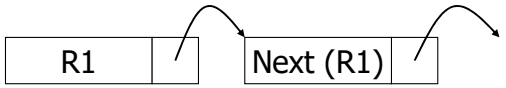
- e.g., to do a merge-join — discussed later

Sequencing Options (1)

(a) Next record physically contiguous

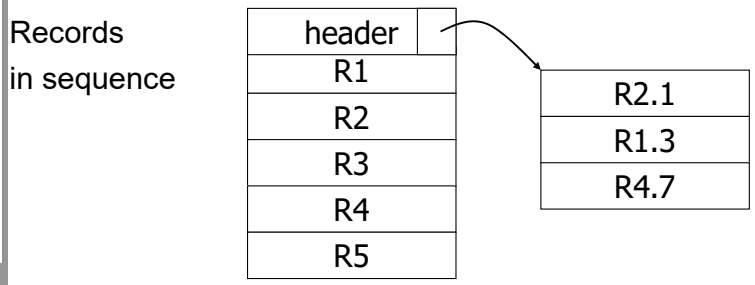


(b) Linked



Sequencing Options (2)

(c) Overflow area



Addressing

How does one refer to records?

Two approaches:

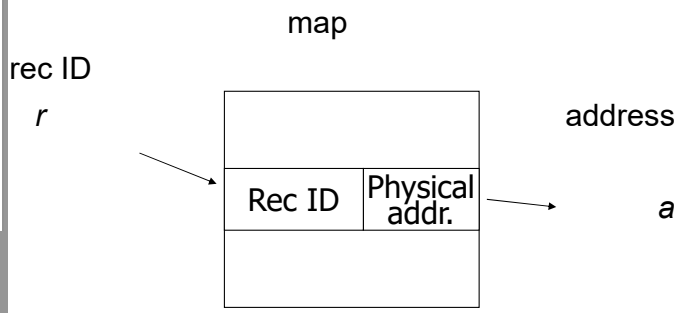
1. Physical: sequence of bytes describing location

For example in case of a HDD: device ID, cylinder #, surface #, block # within track, offset within block (for records).

2. Logical (indirect): a map table associates abstract ID's, perhaps fixed-length character strings, with physical addresses.

Fully Indirect

E.g., Record ID is arbitrary bit string



Addressing (Cont.)

The map table is itself a relation; Fast access is important.

Physical addresses are more efficient.

Logical addresses allow flexibility:

- records can move or be deleted without dangling pointers.

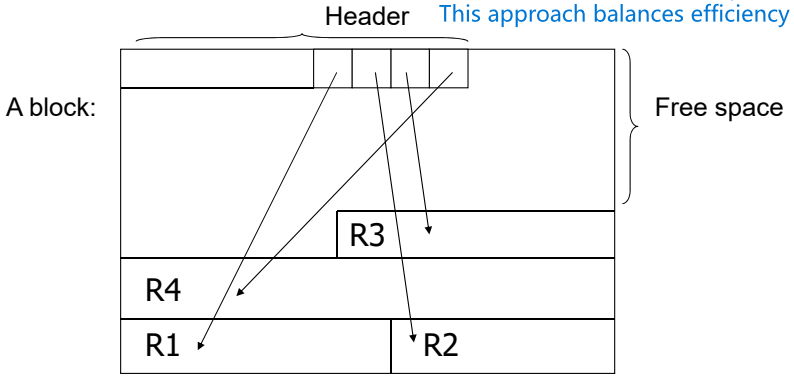
Common compromise: physical to block level, table of record offsets within blocks.

- Tuple-Identifier (TID)
- movement within block: references remain unchanged; only table is modified
- movement to different block: original block contains another TID instead of the record
- regular reorganizations necessary

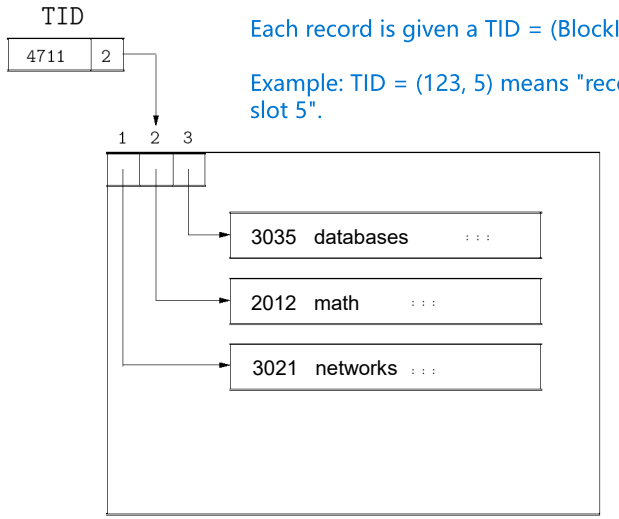
Indirection in Block

- Common Compromise in DBMS

DBMSs usually mix both approaches:
Physical addressing to the block level.
Each record is identified by its block number (fast access).
Logical addressing inside the block.
Within a block, a slot table (offset table) tells where each record s
This approach balances efficiency and flexibility.



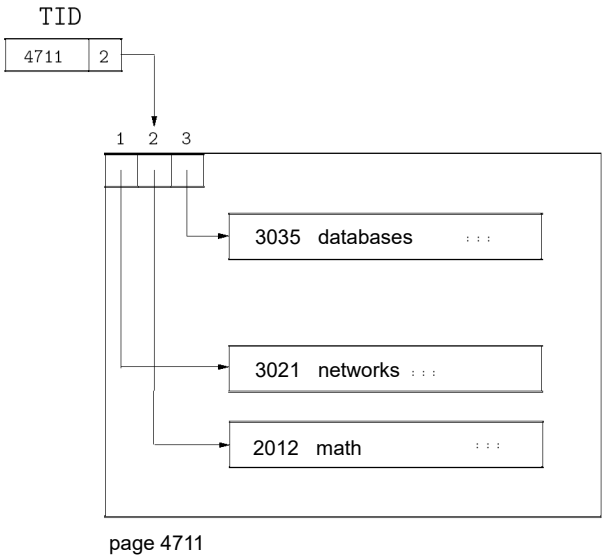
TID Example



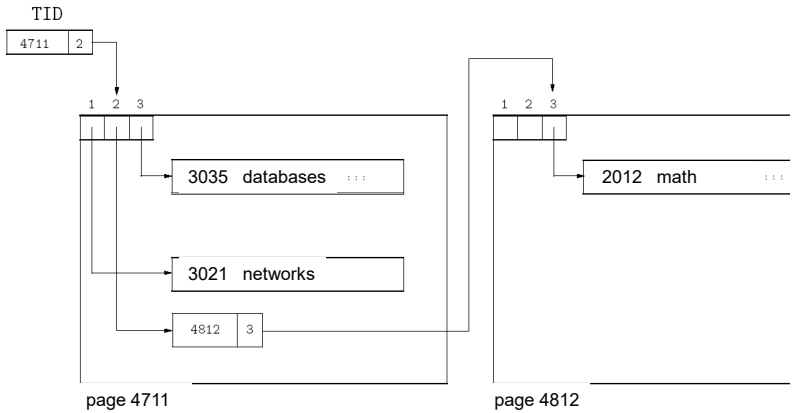
Each record is given a TID = (BlockID, SlotID).

Example: TID = (123, 5) means "record in Block 123, slot 5".

Movement Within Block



Movement to Different Block



Options for Deletion

(a) Immediately reclaim space

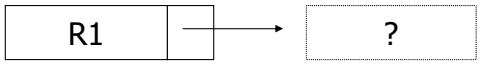
(b) Mark deleted

- Need a way to mark:
  - special characters
  - in map
- May need chain of deleted records (for re-use)

As usual many tradeoffs

- How expensive is it to move valid record to free space for immediate reclaim?
- How much space is wasted?
  - e.g., deleted records, delete fields, free space chains,...

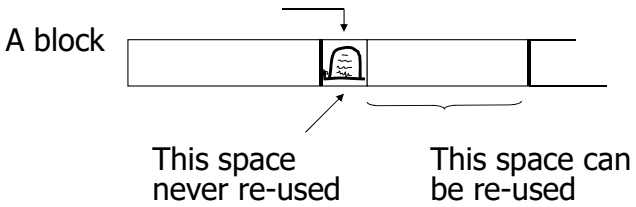
Dangling Pointers



Solution: Tombstones (1)

E.g., leave “MARK” in map or old location

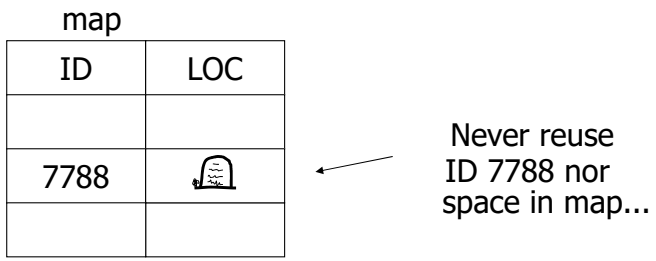
- Physical IDs



Solution: Tombstones (2)

E.g., Leave “MARK” in map or old location

- Logical IDs



Pointer Swizzling

Typical DB structure:

- Data maintained by DBMS, using physical or logical addresses of perhaps 8 bytes.
- Application programs are clients with their own (conventional, virtual-memory) address spaces.

When blocks and records are copied to client's memory, DB addresses can be swizzled = translated to virtualmemory addresses.

- Allows conventional pointer following.
- Especially important in OODBMS, where pointers refer to other objects.

Swizzling Options

1. Never swizzle. Keep a translation table of DB pointers to local pointers; consult table to follow any DB pointer.
   - Problem: time to follow pointers.
2. Automatic swizzling. When a block is copied to memory, replace all its DB pointers by local pointers.
   - Problem: large investment if not too many pointerfollowings occur.
3. Swizzle on demand. When a block is copied to memory, do not translate pointers within the block. If we follow a pointer, translate it the first time.
   - Problem: requires a bit in pointer fields for DB/local, extra decision at each pointer following.

Returning Blocks to Disk

Pointers in the returned block must be unswizzled.

Locate swizzled pointers to block (list has to be managed) and unswizzle.

Index Structures

What is an Index?

An index is a data structure that allows us to directly locate units of data based on certain values

- Not just used for databases: also books can contain an index

Indexes for databases are used to find records that have a particular value for the indexed attribute (the “search key”)

An index has to be created before it can be used

- creation often initiated by the database designer
- cost of maintenance

Different categories exist

- primary / secondary indexes — based on the key being indexed.
- dense / sparse indexes — based on whether an index entry exists for every record or only for certain records.

Sequential Files

Records ordered by search key (may not be "key" in DB sense).

- facilitates queries on the search key

Blocks containing records therefore ordered.

- physically contiguous
- chained

On insert: put record in appropriate block if room.

- Good idea: initialize blocks to be less than full; reorganize periodically if file grows.

If no room in proper block:

- 1. Create new block; insert into proper order if possible (what if blocks are consecutive around a track for efficiency?).
- 2. If not possible, create overflow block, linked from original block.

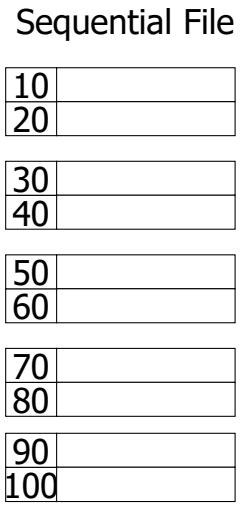
Indexes

Dense Indexes: Pointer to every record of file, ordered by search key.

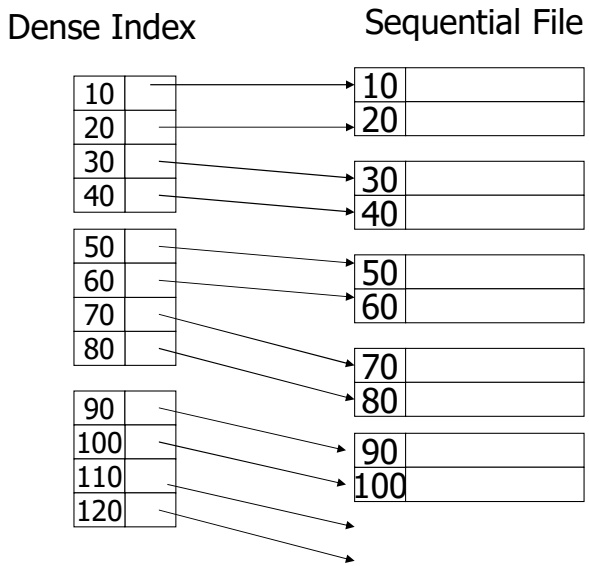
- Can make sense because records may be much bigger than key-pointer pairs.
  - If index requires fewer blocks faster search through index than data file
  - Index might fit in memory, even if data file does not
- Test existence of record without going to data file.

Sparse Indexes: Keypointer pairs for only a subset of records, typically first in each block.

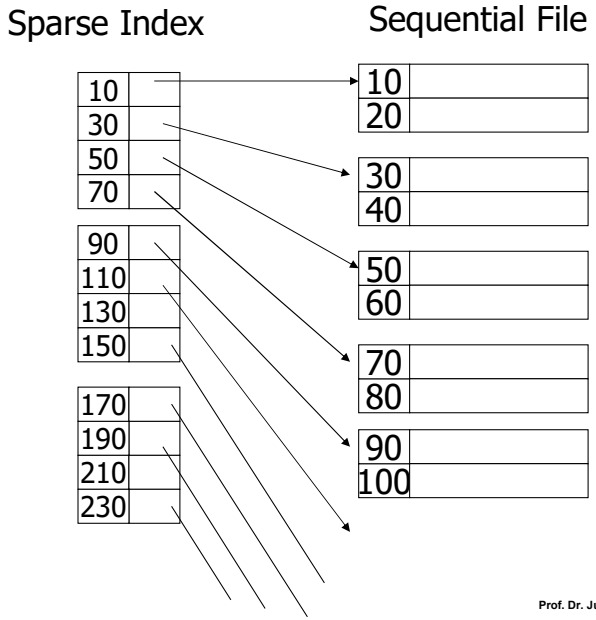
Example: Sequential File



Example: Dense Index



Example: Sparse Index



Sparse vs. Dense Index

Sparse: Less index space per record
can keep more of index in memory

Dense: Can tell if any record exists
without accessing file

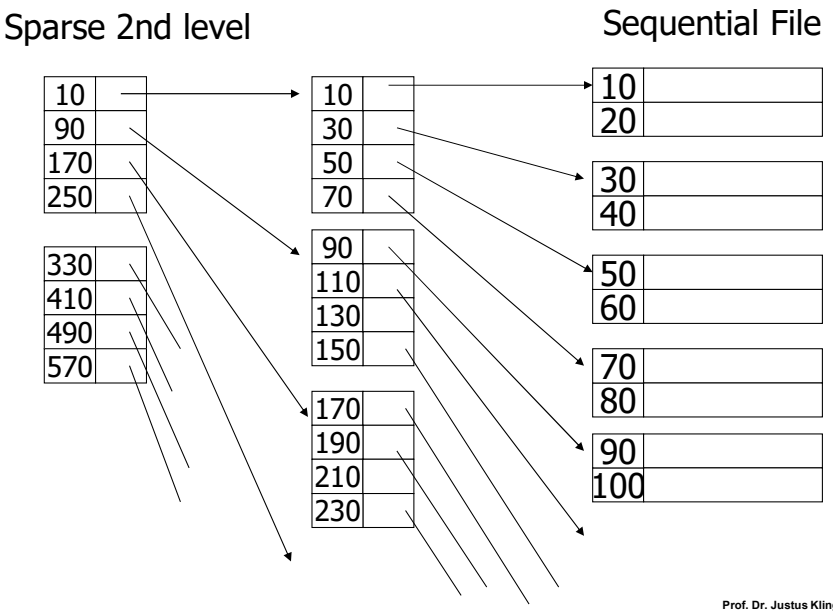
Multiple Levels of Index

A sparse index on a (sparse or dense) index is an option.

Good chance that 2nd or higher level indexes can be housed in main memory, so no additional disk I/O's.

Dense higher level indexes make no sense;

Example: Second Level Index



DB Modifications

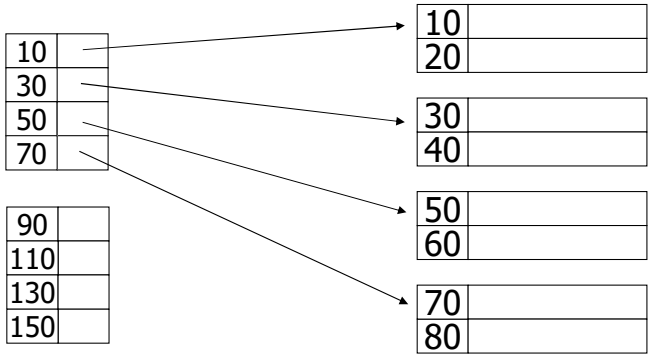
When we insert or delete on the data file, here are the primitive actions we might take:

1. Create or destroy an empty block in the sequence of blocks belonging to the sequential file.
2. Create or destroy an overflow block.
3. Insert a record into a block that has room.
4. Delete a record.
5. Slide a record to an adjacent block.

Effect of Primitive Actions on Index File

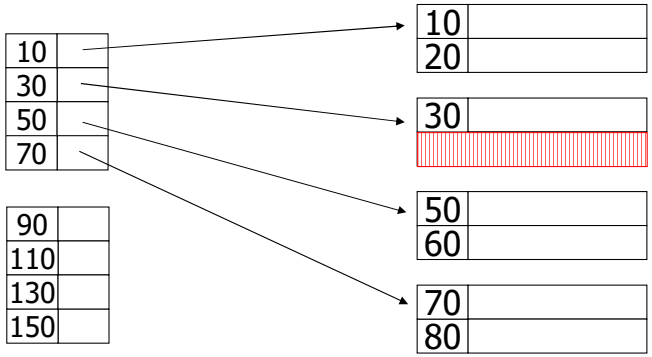
Action	Dense	Sparse
Create/destroy empty overflow block	none	none
Create empty seq. block	none	insert
Destroy empty seq. block	none	delete
Insert record	insert	update(?)
Delete record	delete	update(?)
Slide record	update	update(?)

Deletion from sparse index



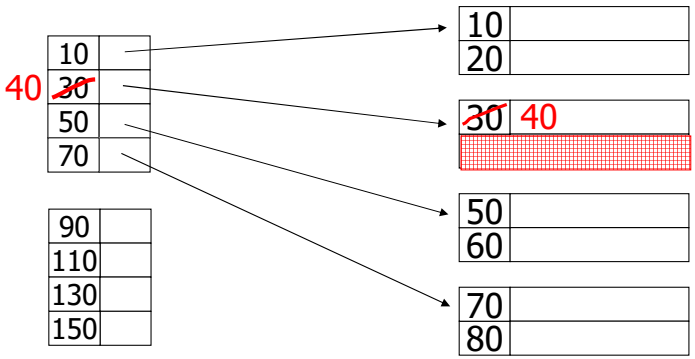
Deletion from sparse index

– delete record 40



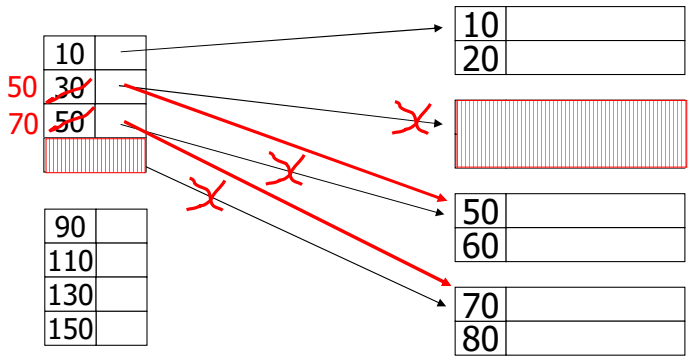
Deletion from sparse index

– delete record 30

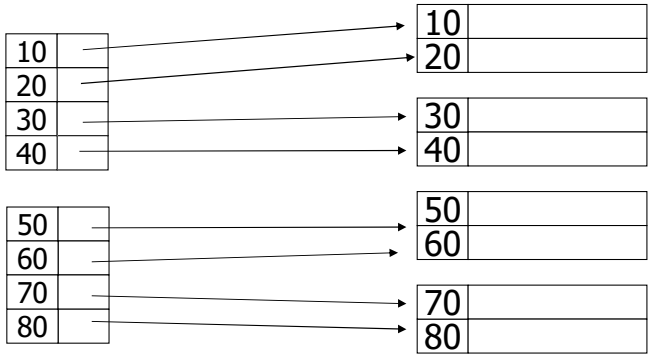


Deletion from sparse index

– delete records 30 & 40

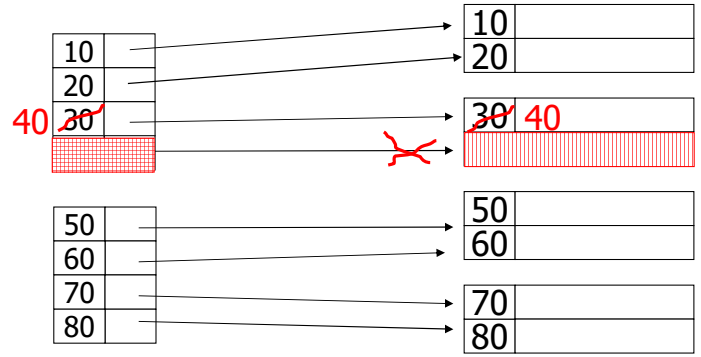


Deletion from dense index

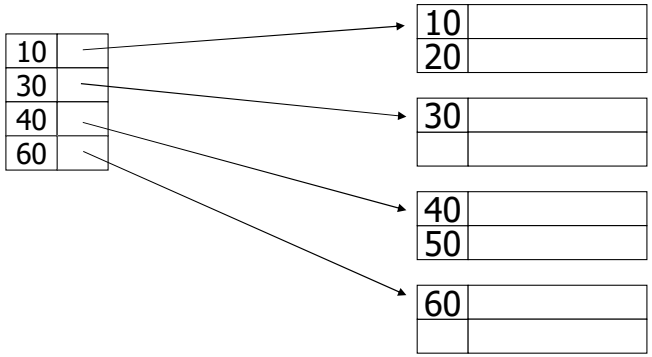


Deletion from dense index

– delete record 30

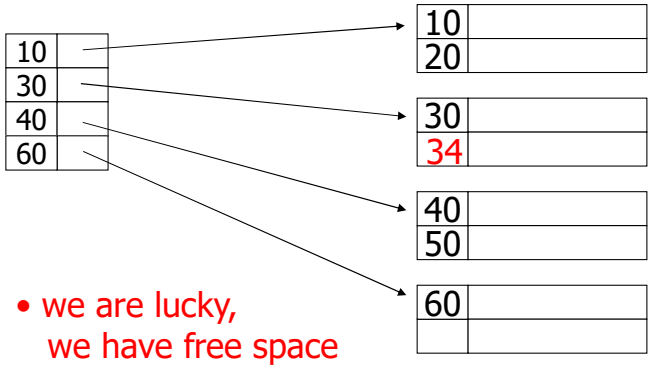


Insertion, sparse index case



Insertion, sparse index case

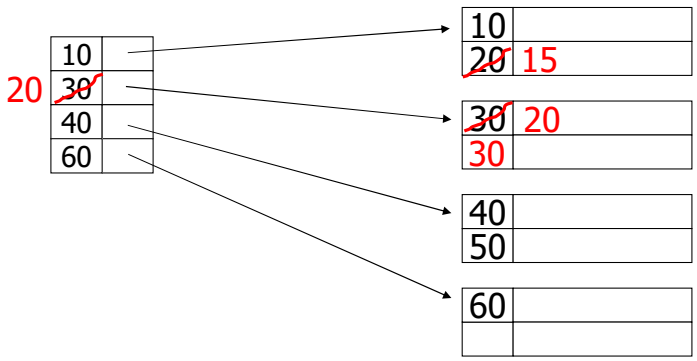
– insert record 34



- we are lucky, we have free space where we need it!

Insertion, sparse index case

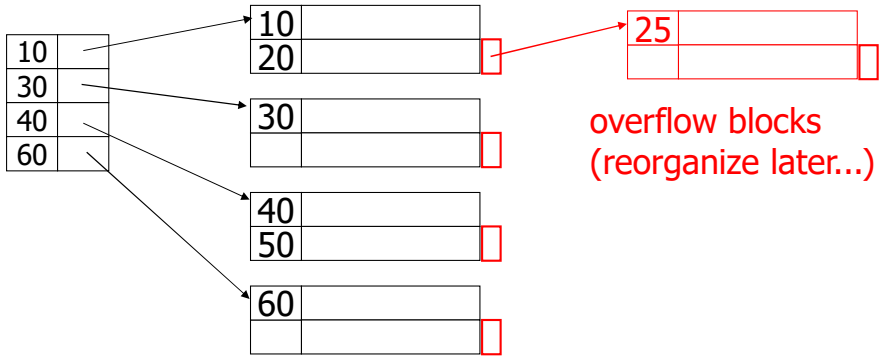
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

Insertion, sparse index case

– insert record 25



Insertion, dense index case

- Similar
- Often more expensive . . .

Dense indexes trade insertion cost for faster search. The extra cost is due to:

Maintaining sorted order

Handling larger index structures

Performing additional I/O when splits happen.

Secondary Indexes

A primary index is an index on a sorted file.

- More general: any index that "controls" the placement of records to be primary, e.g., hash table.

Secondary index = index that does not control placement, surely not on a file sorted by its search key.

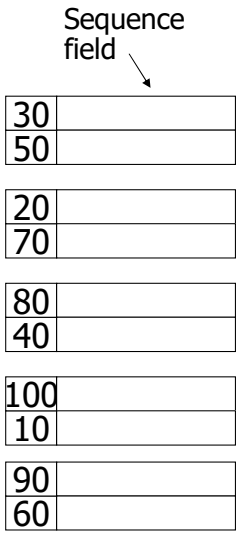
- Sparse, secondary index makes no sense.
- Usually, search key is not a "key"

Multiple Levels:

- Lowest level is dense
- Other levels are sparse

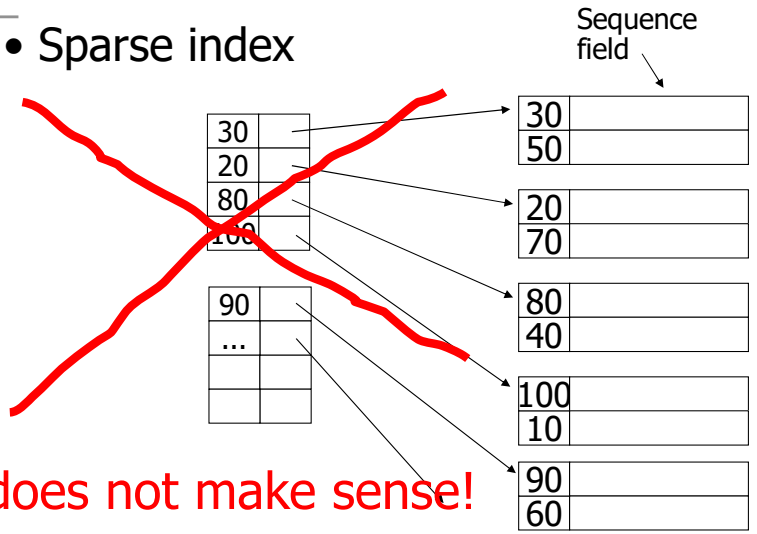
A secondary index must be dense because:
Data is not ordered by the secondary key.
You need a pointer for every record to locate them accurately.
If you keep only one entry per block, you can't guarantee finding all rec with a given secondary key.
You would miss records because there is no contiguous structure.

Secondary Indexes



Secondary Indexes

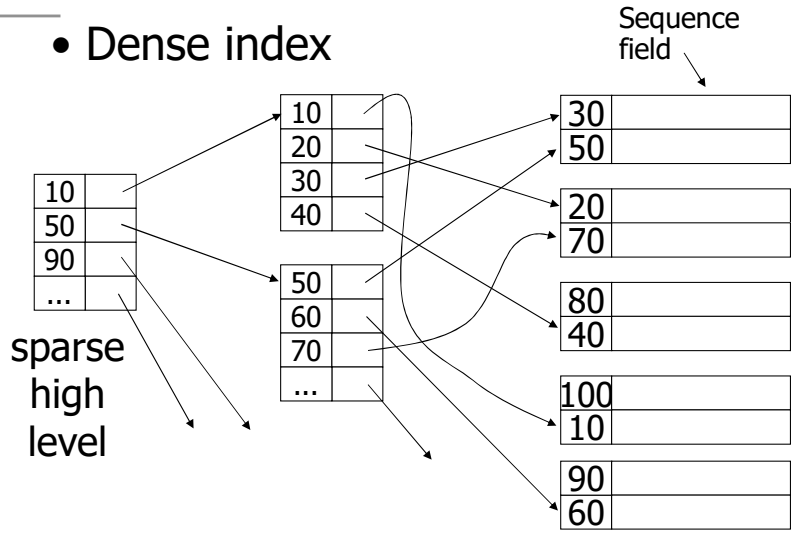
- Sparse index



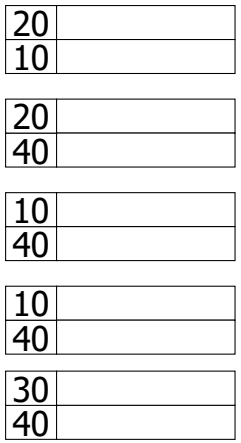
does not make sense!

Secondary Indexes

- Dense index

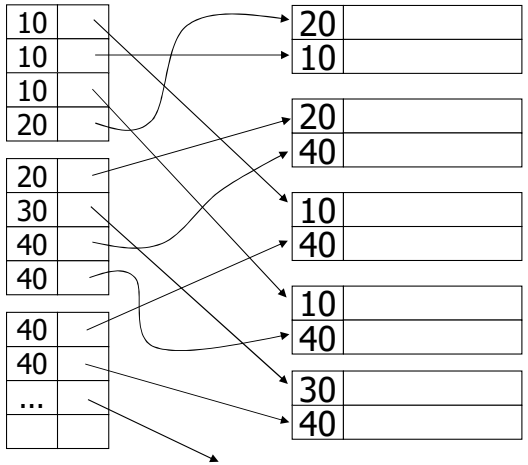


Duplicate values & secondary indexes



Duplicate values & secondary indexes

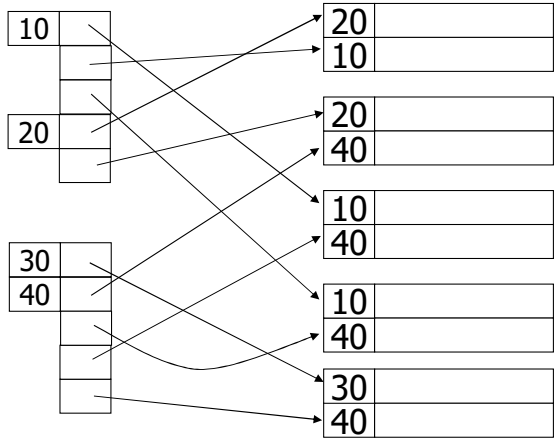
one option...



Problem:
excess overhead!
- disk space
- search time

Duplicate values & secondary indexes

another option...



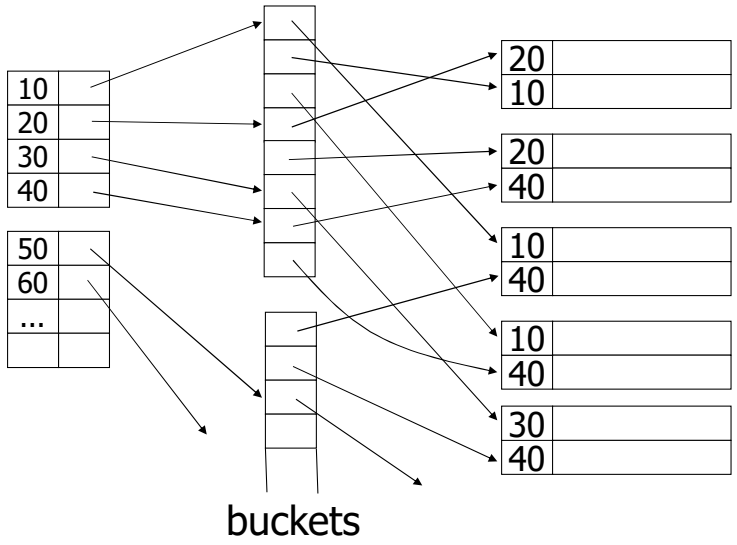
Problem:
variable size records in index!

Indirect Buckets

To avoid repeating keys in index, use a level of indirection, called buckets.

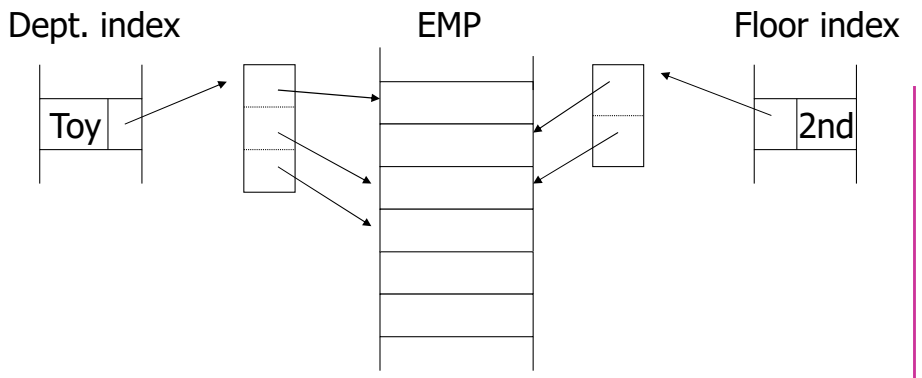
- Additional advantage: allows intersection of sets of records without looking at records themselves.

Duplicate values & secondary indexes



Indirect Buckets

Query: Get employees in
(Toy Dept) ∧ (2nd floor)



→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's

Assessment of Conventional Indexes

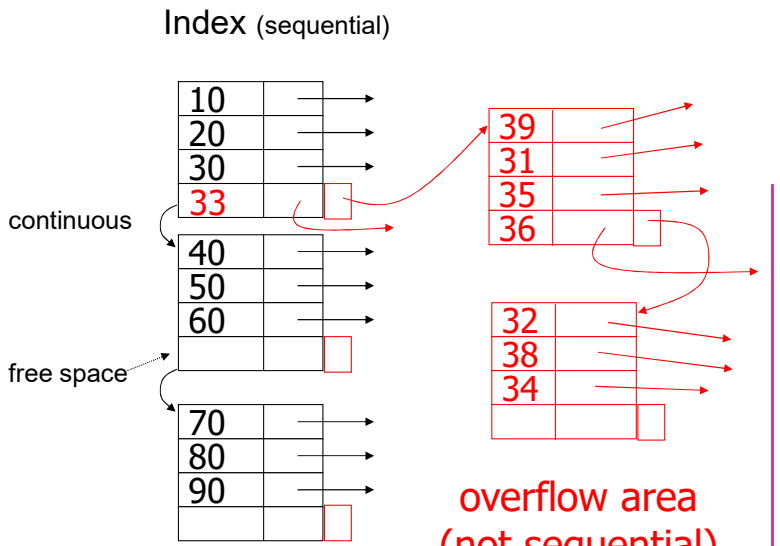
Advantage:

- Simple
- Index is sequential file good for scans

Disadvantage:

- Inserts expensive, and/or
- Lose sequentiality & balance

Example



B-Trees

B-Trees

Generalizes multilevel index.

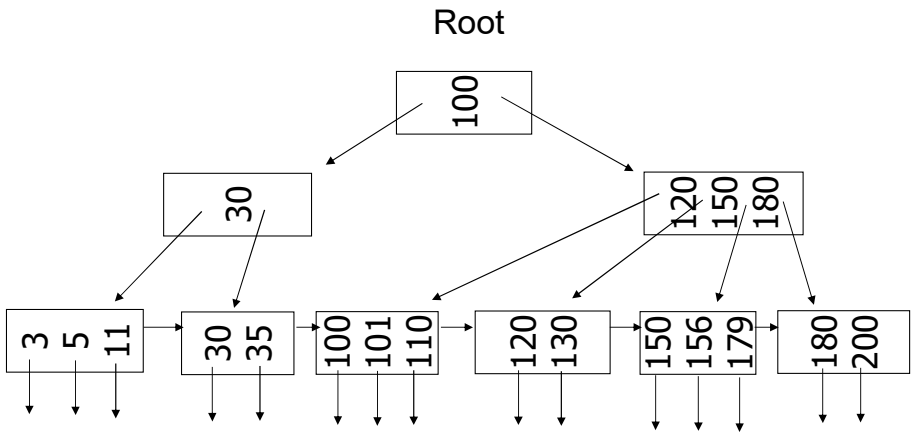
Number of levels varies with size of data file, but is often 3.

Different variants, we start with B+-trees.

Useful for primary, secondary indexes, primary keys, nonkeys.

Each node in the tree represents a block.

B+Tree Example



Nodes of B+ Tree

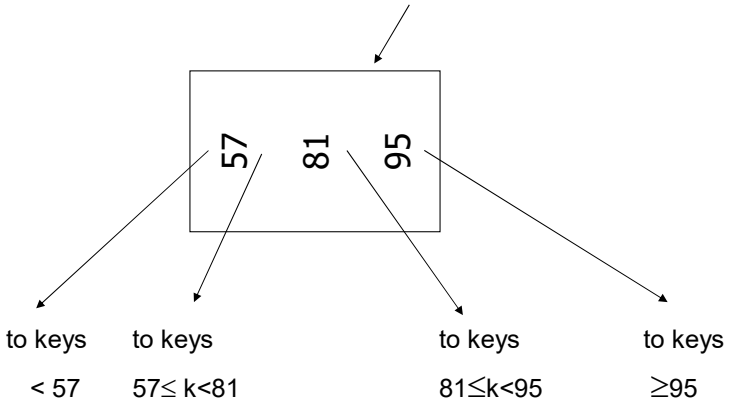
Leaves

- One pointer to next leaf.
- keypointer pairs for records of data file.
- At least half of these (round up) occupied.

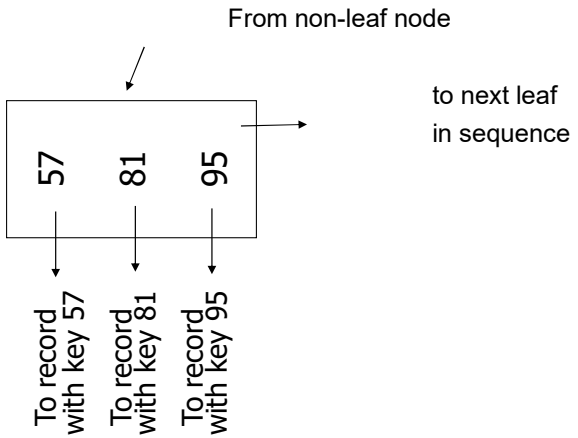
Interior Nodes

- k keys form the divisions among k+1 subtrees.
- Key i is least key reachable from (i + 1)st child.

Sample non-leaf



Sample Leaf Node



Don't want nodes to be too empty

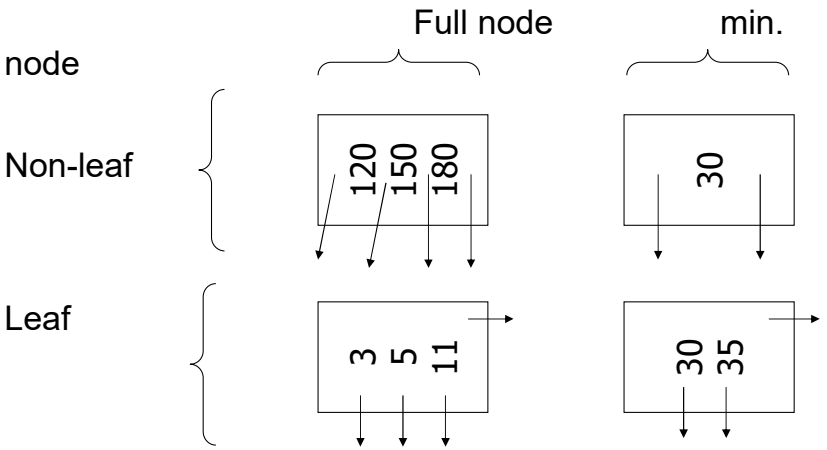
Trees have an order that determines the maximal number of keys in a node

Use in a tree of order n at least

Non-leaf:	$\lceil (n+1)/2 \rceil$	pointers to children
Leaf:	$\lfloor (n+1)/2 \rfloor$	pointers to records

Root is a special Case

n=3



B+ Tree rules (Tree of order n)

(1) All leaves at same lowest level (balanced tree)

(2) Pointers in leaves point to records except for "sequence pointer"

(3) Number of pointers/keys for B+ tree (except for sequence pointers)

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1 (if leaf)	1

Lookup

Lookup in B+ Tree

- Start at root.
- Until you reach a leaf, follow the pointer that could lead to the key you want.
- Search that leaf (and leaves to the right if duplicates are possible).

B+ Tree Insertion

Search for the key being inserted.

If there is room for another key-pointer pair at that leaf, insert there.

If no room, split leaf.

- Split of leaf results in insert of key-pointer pair at level above.
  - key is **copied** to level above
- Thus, recursive splitting all the way up the tree is possible.
  - split of non-leaf results in **moving** one key to level above
- Convention: If the number of keys in the two nodes resulting from the split is uneven, put one more key in the left node. Otherwise: both nodes get the same number of keys

Examples for Insert into B+ Tree

(a) simple case
- space available in leaf

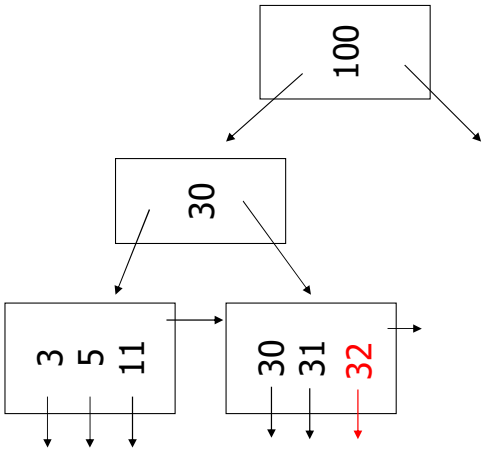
(b) leaf overflow

(c) non-leaf overflow

(d) new root

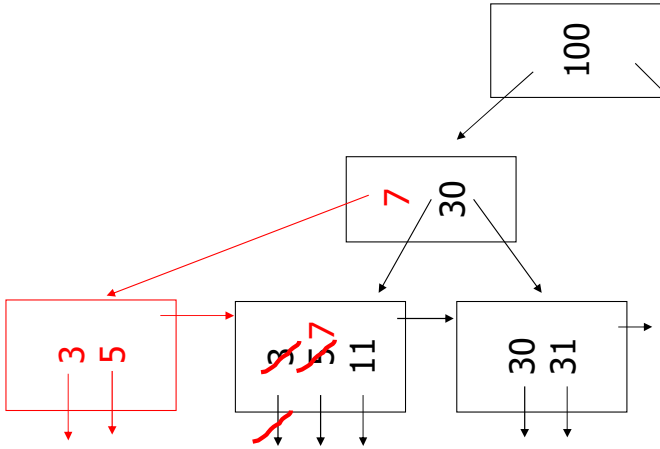
(a) Insert key = 32

n=3



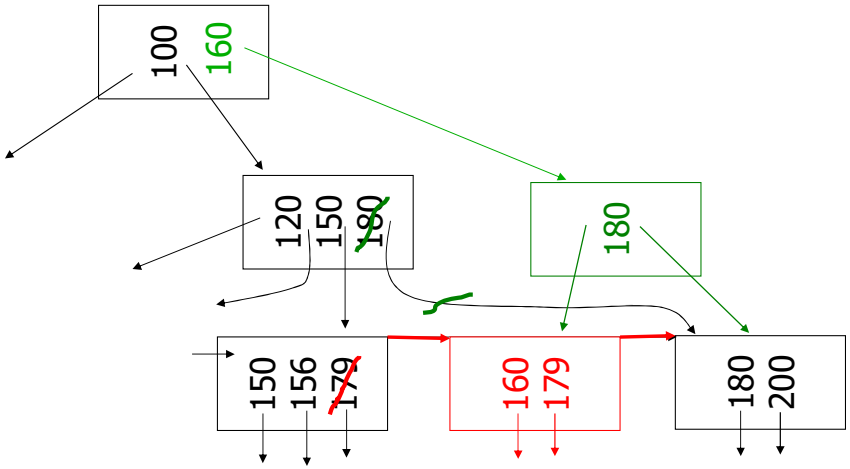
(b) Insert key = 7

n=3



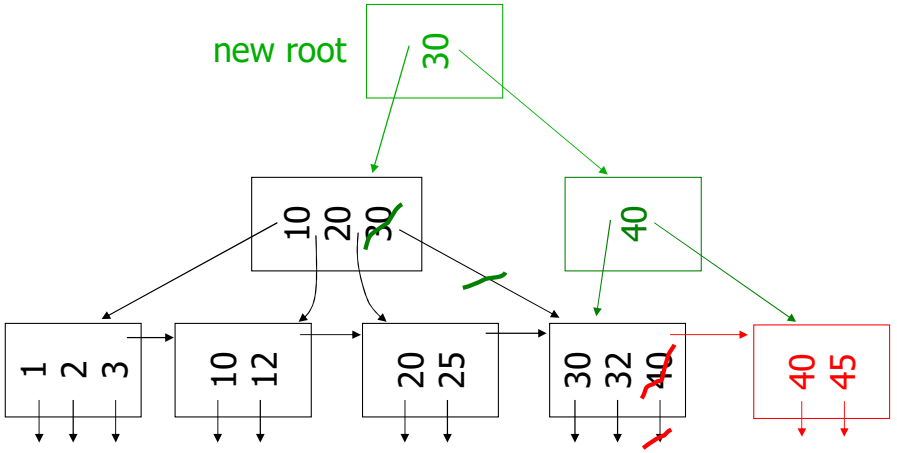
(c) Insert key = 160

n=3



(d) New root, insert 45

n=3



B+ Tree Deletion

Search for key being deleted; If found, delete from the leaf.

If the lower limit on occupancy is violated:

- First look for an adjacent sibling that is above lower limit; transfer a key-pointer pair from that node (and update parent).
  - Convention: If you have the choice, use left sibling
  - A transfer between non-leaves involves a key in the parent and also results in the transfer of a child
- If none, then there must be two adjacent leaves, one at minimum, one below minimum. Just enough to merge nodes.
  - Convention: If you have the choice, use left sibling
  - A merge is the opposite of a split: delete key in parent when merging leaves; move key from parent into merged node for non-leaves
- Merger looks like delete above, so recursive deletion possible.
- Again, make sure keys are adjusted above.

Sometimes, it is OK to allow a B+ Tree leaf to become subminimum. But we handle underflows!!!

Deletion from B+ Tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

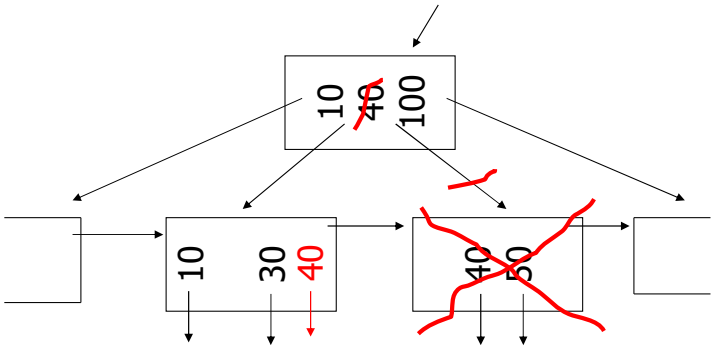
(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

- Delete 50

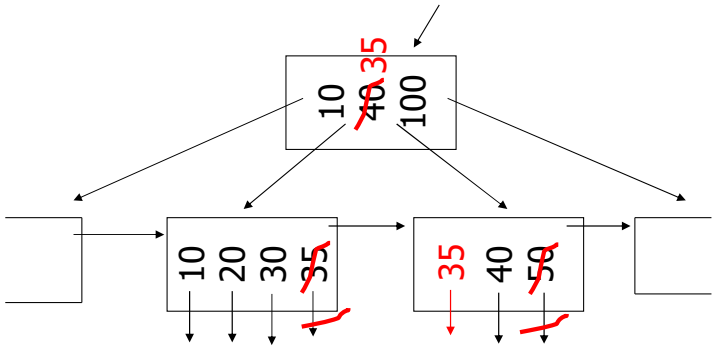
n=4



(c) Redistribute keys

- Delete 50

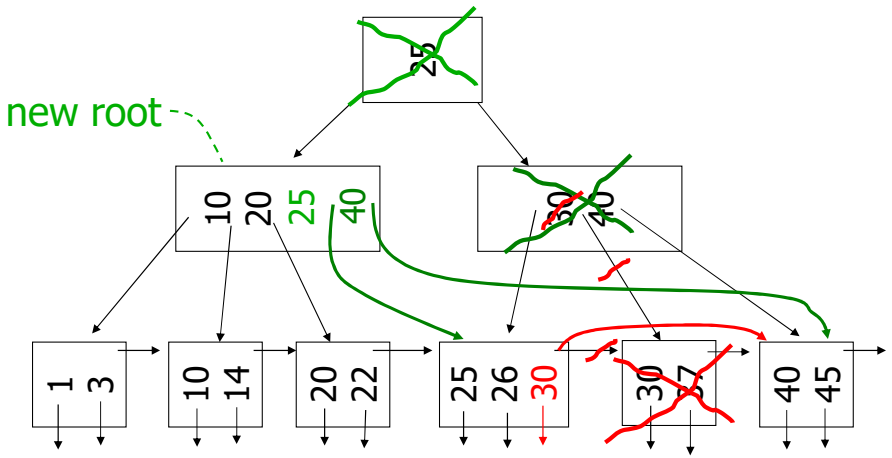
n=4



(d) Non-leaf coalese

- Delete 37

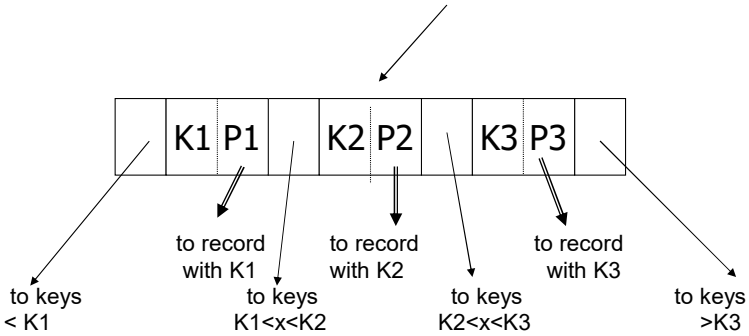
n=4



Variation on B+ Tree: B Tree (no +)

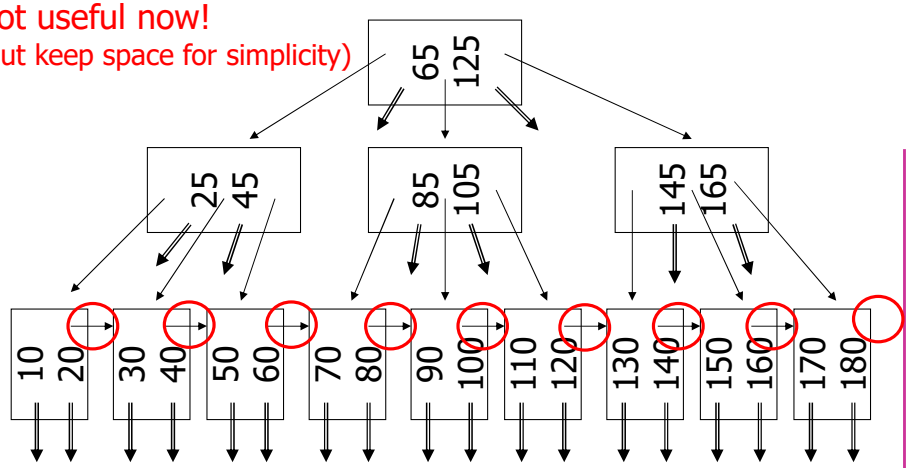
Idea:

- Avoid duplicate keys
- Have record pointers in non-leaf nodes



B Tree example (n=2)

sequence pointers
not useful now!
(but keep space for simplicity)



B Tree operations

Ideas for search, insertion and deletion are similar to B+-trees

Main difference: We do not have to keep all keys in leaves

Consequence: When splitting a leaf, we **move** one key to the level above (also **move** when merging leaves)

Results:

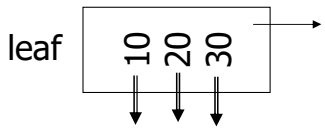
- split / merge for leaves are performed like the corresponding operations for non-leaves
- the minimum number of keys (and pointers) in a leaf is the same as for a non-leaf: $\lceil (n+1)/2 \rceil - 1$ keys

Deletion:

- when deleting a key that is in a non-leaf: replace the key with the next larger key in the tree
- handling of underflows always starts from a leaf

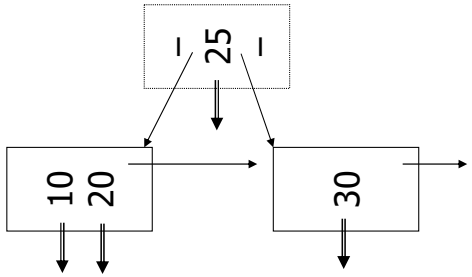
Example: Insert

Insert record with key = 25



n=3

Afterwards:



Comparison

☺ B-trees have faster lookup for keys in internal nodes than B+-trees

☹ in real implementations of a B-trees, non-leaf nodes can store a smaller number of keys compared to B+-trees due to the additional pointers

☹Therefore, in B-trees the height of a tree for a particular number of keys can be larger compared to a B+-tree

➨ B+-trees are usually preferred!

Lookup for B+-tree is actually better!!

Example

- Pointers 4 bytes
- Keys 4 bytes
- Blocks 100 bytes
- Look at full 2 level tree

B tree

100/(4key byte+4 byte point to recor+4 byte pointer to node)
100/12 = 8 keys

Root has 8 keys + 8 record pointers + 9 child pointers
= 8x4 + 8x4 + 9x4 = 100 bytes

Each of 9 childs: 12 rec. pointers (+12 keys)
= 12x(4+4) = 96 bytes

2-level B-tree, Max # records =
12x9 + 8 = 116

B+tree

Root has 12 keys + 13 child pointers

= 12x4 + 13x4 = 100 bytes

Each of 13 childs: 12 rec. ptrs (+12 keys)

= 12x(4 +4) + 4 = 100 bytes

2-level B+tree, Max # records

= 13x12 = 156

Conclusion:

- For fixed block size a B+ tree is better
- each node can store more keys and pointers to child nodes

Hashing

Hash Tables

Hash function h: search key -> [0, ..., B-1].

Buckets are blocks, numbered [0, ..., B-1].

General idea: If a record with search key K exists, then it must be in bucket h(K).

- Cuts search down by a factor of B.
- One disk I/O if there is only one block per bucket.

Hash Table Operations

HashTable Lookup

- For record(s) with search key K, compute h(K); search that bucket.

HashTable Insertion

- Put in bucket h(K) if it fits; otherwise create an overflow block.
- Overflow block(s) are part of bucket.

HashTable Deletion

- Compute h(K); search bucket for record(s) with key K and delete entry

Example with 2 Records/Bucket

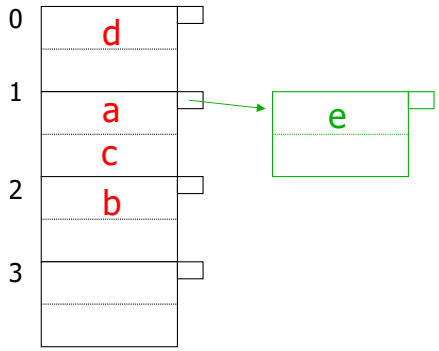
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



h(e) = 1

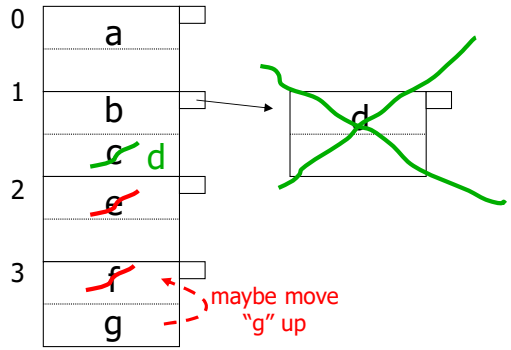
Example: Deletion

Delete:

e

f

c



How full should a Block be?

Try to keep space utilization

between 50% and 80%

Utilization = $\frac{\text{\# keys used}}{\text{total \# keys that fit}}$

If < 50%, wasting space

If > 80%, overflows significant

depends on how good hash function is and on # keys/bucket r/n

How do we cope with growth?

Overflows and reorganizations

Dynamic hashing

Extensible

Linear

Dynamic Hashing Framework

Hash function h produces a sequence of bits.

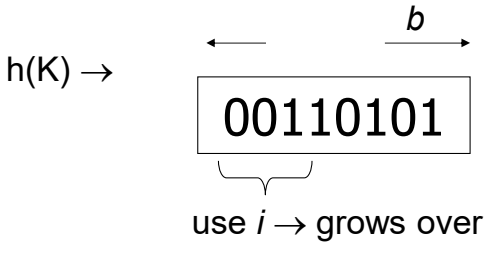
Only some of the bits are used at any time to determine placement of keys in buckets.

Extensible Hashing

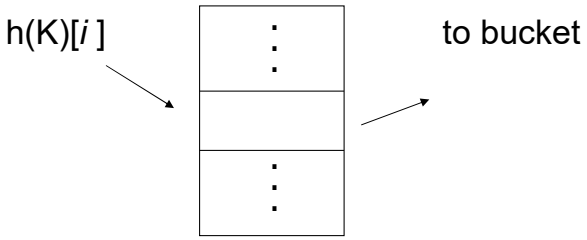
- Keep parameter i = number of bits from the beginning of h(K) determine the bucket.
- Bucket array now = pointers to blocks.
- A block can serve as several buckets.
- For each block, a parameter j <= i tells how many bits of h(K) determine membership in the block.
- I.e., a block represents 2^{i-j} buckets that share the first j bits of their number.

Extensible hashing: two ideas

(a) Use *i* of *b* bits output by hash function



(b) Use directory



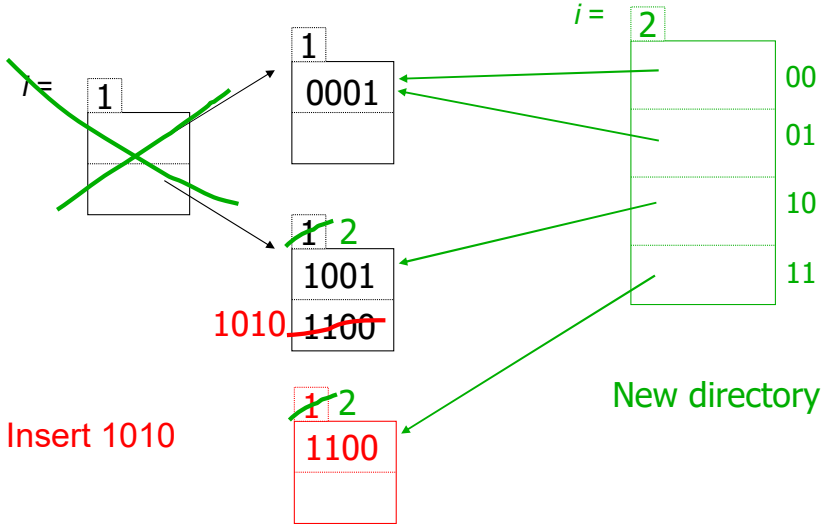
Extensible Hashtable Insert

If record with key K fits in the block pointed to by h(K), put it there.

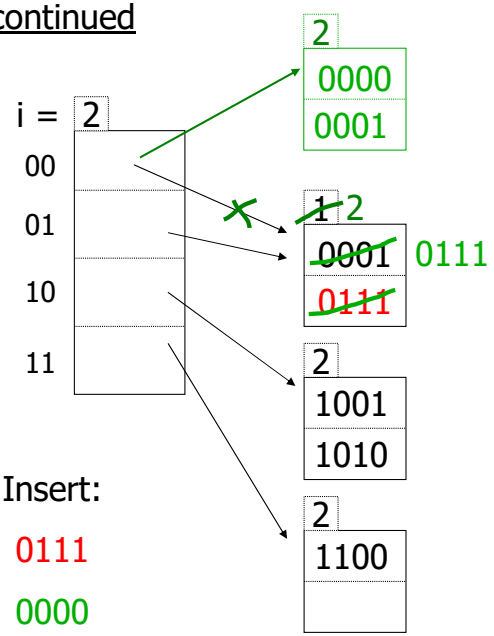
If not, let this block represent j bits.

- Case 1: j < i: Split block according to (j + 1)st bit; set j := j + 1.
- Case 2: j = i: Set i := i + 1; split bucket array; proceed as in (1).

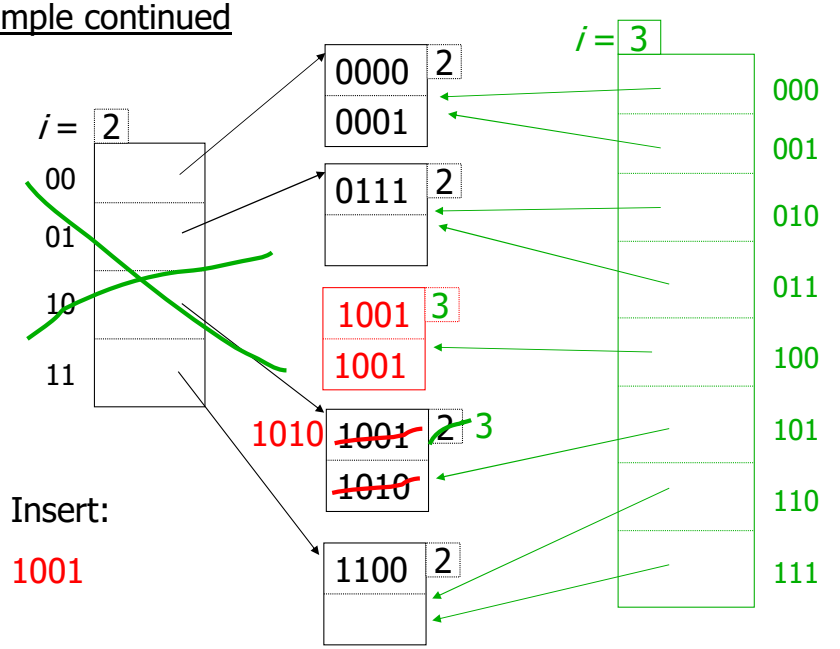
Example: h(k) is 4 bits; 2 records/block



Example continued



Example continued



Summary Extensible Hashing

(+) Can handle growing files
- with less wasted space
- with no full reorganizations

(-) Indirection

(Not bad if directory in memory)

(-) Directory doubles in size

Linear Hashing

Use i bits from right (loworder) end of h(K).

Buckets numbered [0, ..., n-1], where $2^{i-1} < n <= 2^i$.

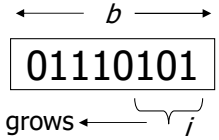
Let the last i bits of h(K) be $m = (a_1 a_2 ... a_i)$.

- 1. If m < n, then record belongs in bucket m.
- 2. If $n <= m < 2^i$, then record belongs in bucket $m - 2^{i-1}$.

Difference to Extensible Hashing

Two ideas:

(a) Use *i* low order bits of hash



(b) File grows linearly

Utilization Limit: A threshold is set for the utilization of the hash table.
Adding a Bucket (n := n + 1): When an insertion causes the utilization to exceed the limit, a new bucket is added to the hash table. This increases the total number of buckets (n).

Linear HashTable Insert

Utilization is defined as the total number of hash entries, divided by the number of possible entries in the primary blocks of hash buckets (i.e., not counting the capacity of overflow blocks!)

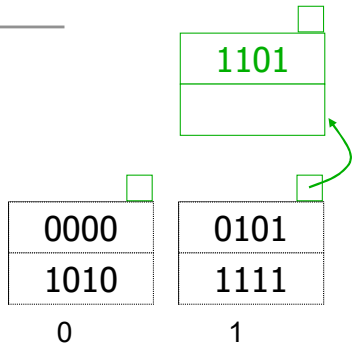
Pick an upper limit on utilization for the hash-table

If an insertion exceeds this utilization limit, add a bucket, i.e., set n := n + 1.

- If new n is $2^i + 1$, set i := i + 1. No change in bucket numbers needed --- just imagine a leading 0.
- Need to split bucket n - 2^{i-1} because there is now a bucket numbered (old) n.

Note, that the bucket that is split need not be one with a large number of records!

Example *b*=4 bits, *i* =1, 2 records/block



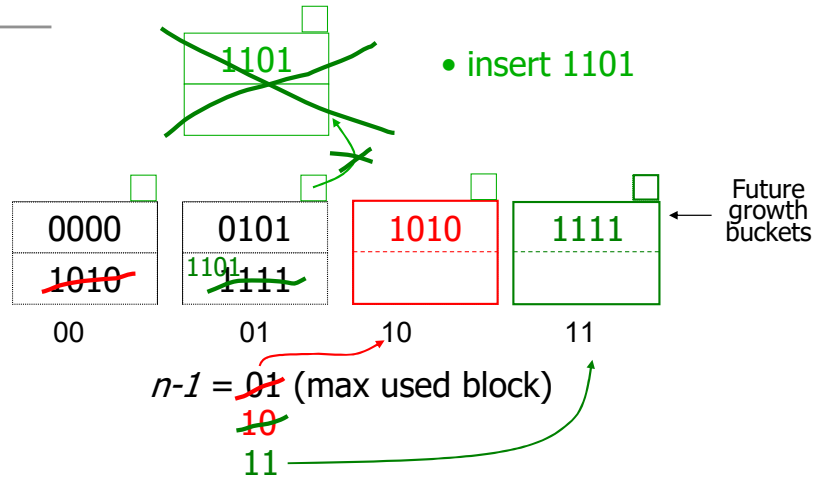
- insert 1101
- can have overflow chains!

n-1 = 1 (max used block)

Rule

If h(k)[*i*] ≤ *n-1*, then
look at bucket h(k)[i]
else, look at bucket h(k)[*i*] - 2^{i-1}

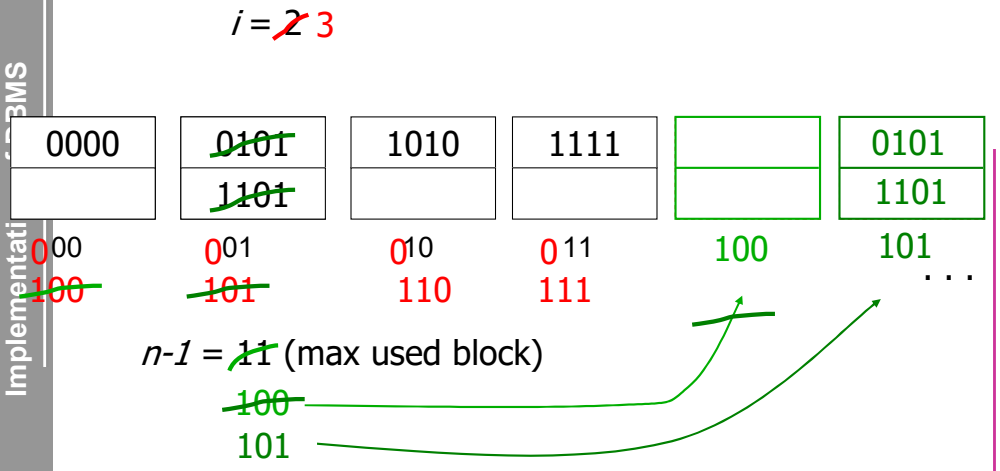
Example *b*=4 bits, *i* =2, 2 records/block



- insert 1101

Growth:
i has increased to 2, indicating that the hash table has grown.
New blocks (1010 and 1111) have been added.
The maximum used block is now n-1 = 01 (in binary).

Example Continued: How to grow beyond this?



Summary Linear Hashing

⊕ Can handle growing files
- with less wasted space
- with no full reorganizations

⊕ No indirection like extensible hashing

⊖ Can still have overflow chains
split might not improve situation for full buckets

Indexing vs Hashing

Hashing good for queries looking for a particular search key

e.g., SELECT …
FROM R
WHERE R.A = 5

Classical indexing and B-Trees good for
Range Searches:

e.g., SELECT
FROM R
WHERE R.A > 5

Digital Trees

Digital Trees

Digital Trees branch based on the letters of the used alphabet

Can become unbalanced

Variants

- Tries (from information re**trie**val)
- Patricia-Trees
- Prefix-Trees

Has its origin in information retrieval

- become popular for database implementation in case of XML-DBMS
  - elements are the alphabet

Trie

Store strings in a tree with an edge for each of its characters.

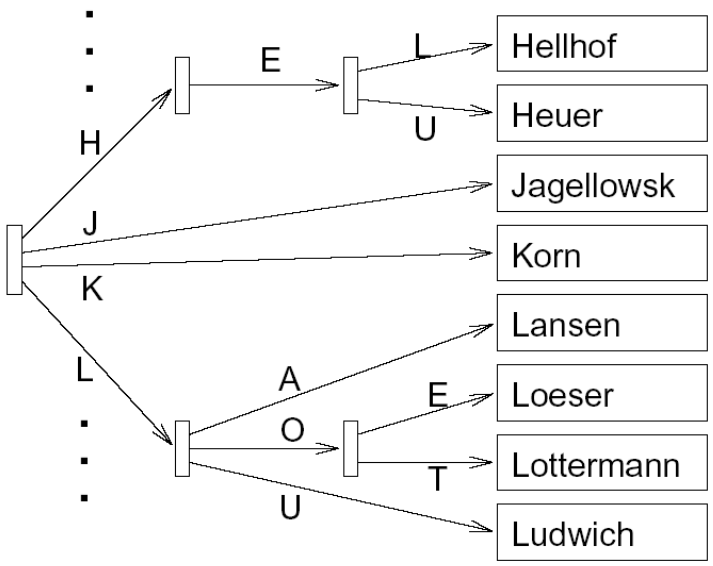
Strings with same prefix share edges in higher tree levels.

The path from the root to a node n describes the word stored in n.

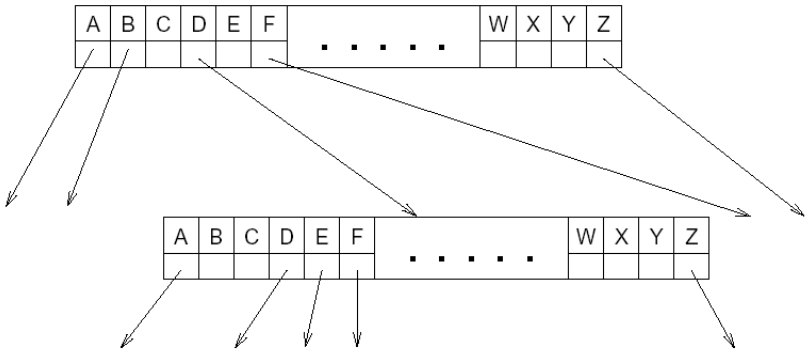
Each node is linked to the results for its corresponding word.

Note, that we have to do up to l steps, where l is the length of the search key.

Example Trie



Nodes of a Trie



Patricia Tree

Problems of a Trie

- Long common sub-words
- Letters that do not occur
- Very unbalanced

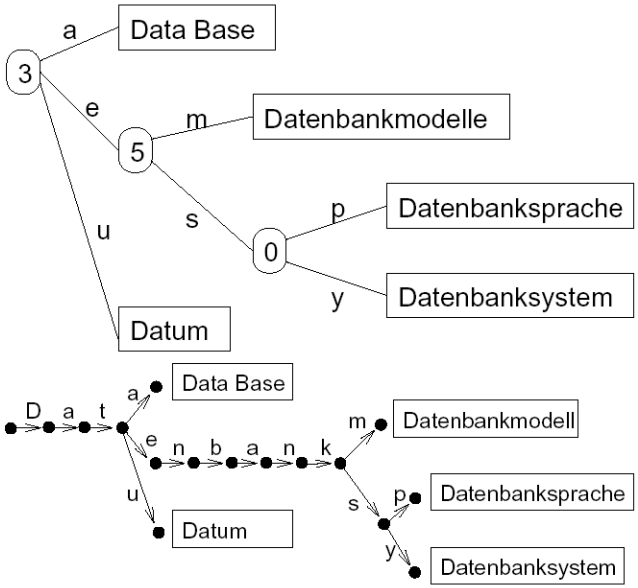
Solution to the first problem: Patricia Trees

- Practical Algorithm To Retrieve Information Coded In Alphanumeric
- Principle: Sub-words that are not needed can be skipped
  - we store the number of letters to be skipped
- We can only determine whether a key exists, when we reach a leaf

Advantages

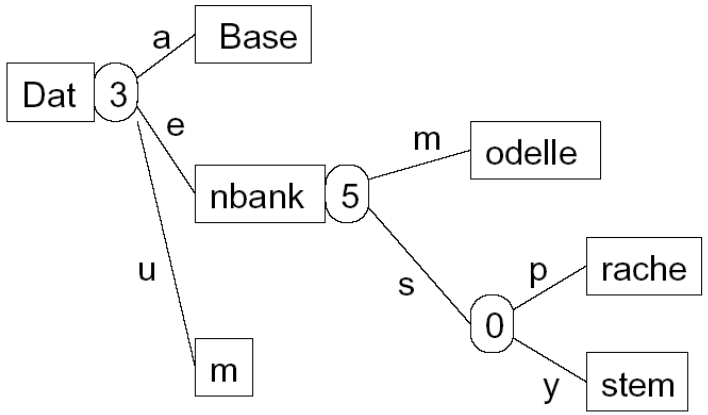
- Patricia trees are very space and CPU efficient.
- With their inherent “compression”, patricia trees grow slowly, even when inserting large strings

Patricia Tree vs. Trie



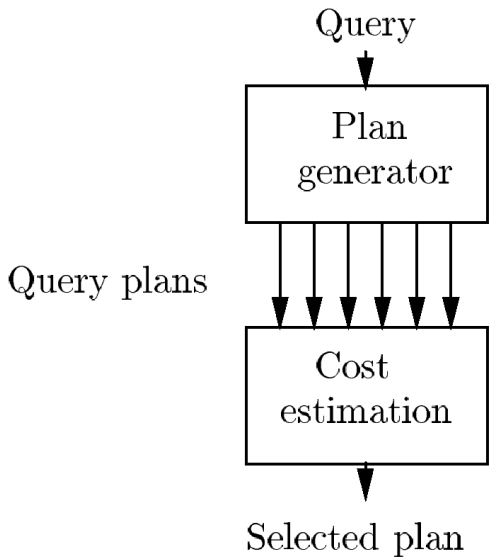
Prefix Tree

Similar to Patricia Tree, but we also store the skipped sub-words in the nodes



Query Processing

Overview Query Processing



Query Plans

Choose operations, e.g., σ, $\bowtie$

Order operations.

Detailed strategy of operations, e.g.:

- Join method.
- Pipelining: consume result of one operation by another, to avoid temporary storage on disk.
- Use of indexes?
- Sort intermediate results?

We focus on relational systems

Example

Select B,D

From R,S

Where R.A = “c” AND S.E = 2 AND R.C=S.C

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3

Answer

B	D
2	x

How Do We Execute the Query?

One idea

- Do Cartesian product
- Select tuples
- Do projection

RXS

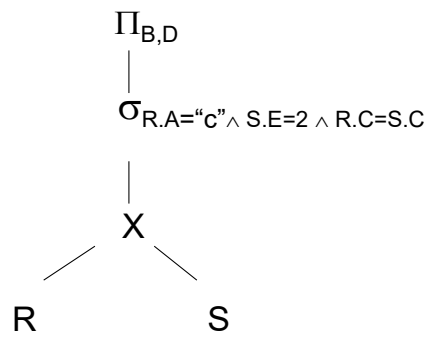
R.A	R.B	R.C	S.C	S.D	S.E
a	1	10	10	x	2
a	1	10	20	y	2
.					
.					
C	2	10	10	x	2
.					
.					

Bingo!

Got one...

Relational Algebra to Describe Plans

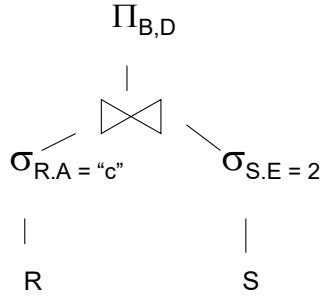
Ex: Plan I



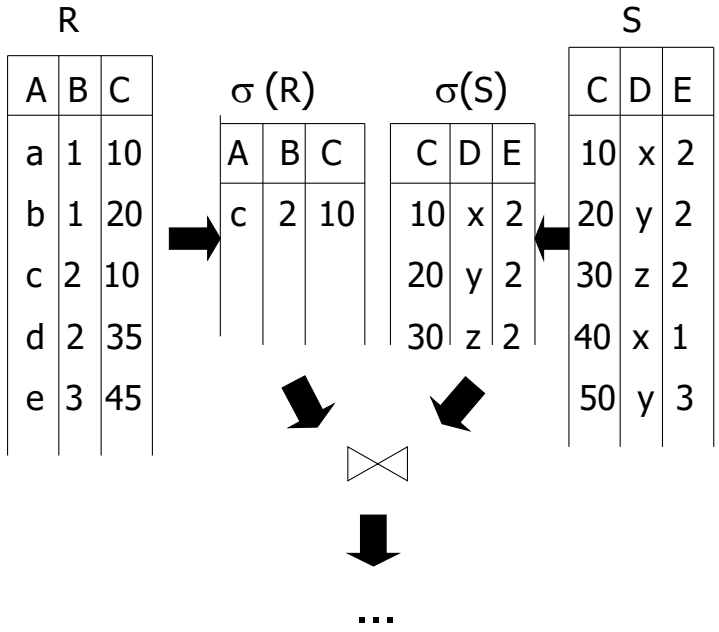
OR: $\Pi_{B,D}[\sigma_{R.A=\text{“c”} \wedge S.E=2 \wedge R.C = S.C}(R \times S)]$

Another Plan

Plan II



natural join



Plan III

Use R.A and S.C Indexes

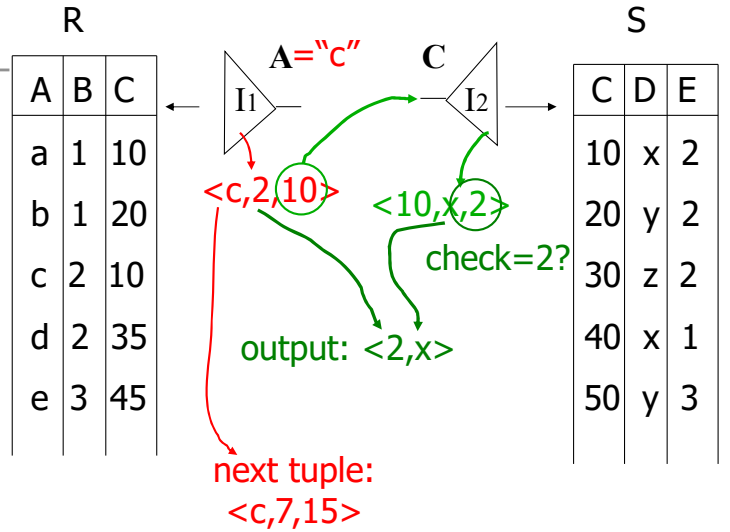
(1) Use R.A index to select R tuples with R.A = “c”

(2) For each R.C value found, use S.C index to find matching tuples

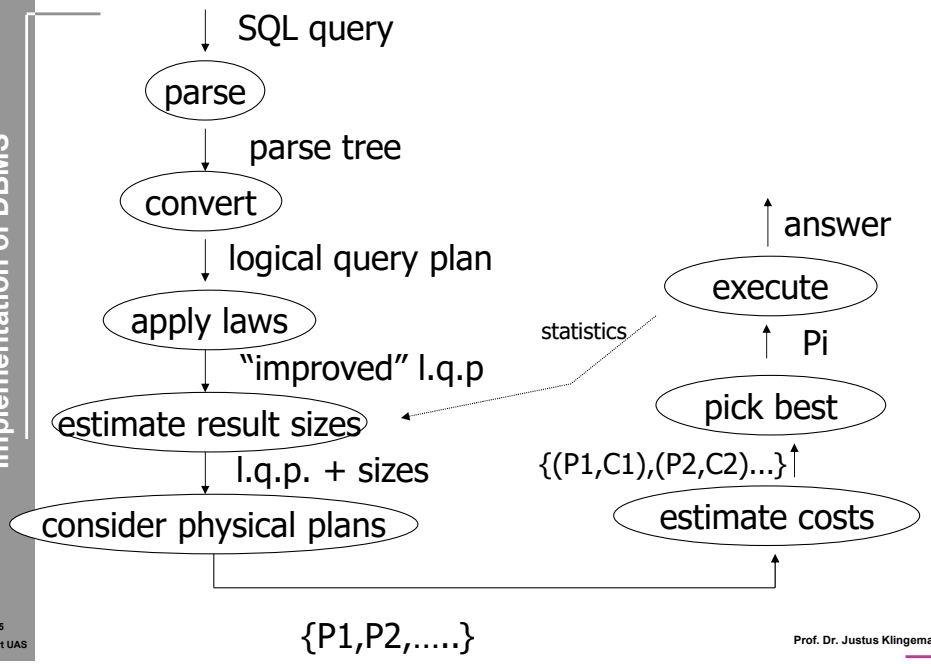
(3) Eliminate S tuples with S.E ≠ 2

(4) Join matching R,S tuples,

(5) Project B,D attributes and place in result



Overview of Query Optimization

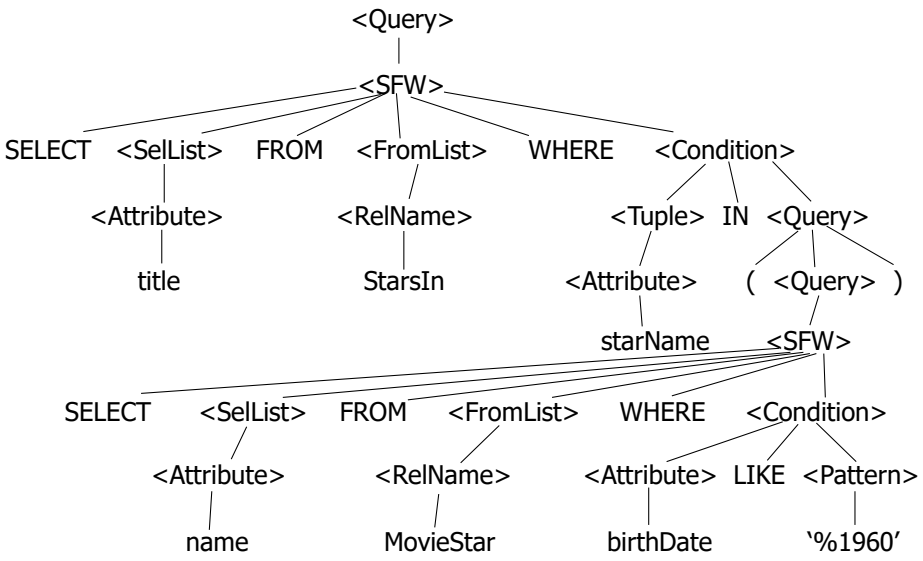


Example: SQL Query

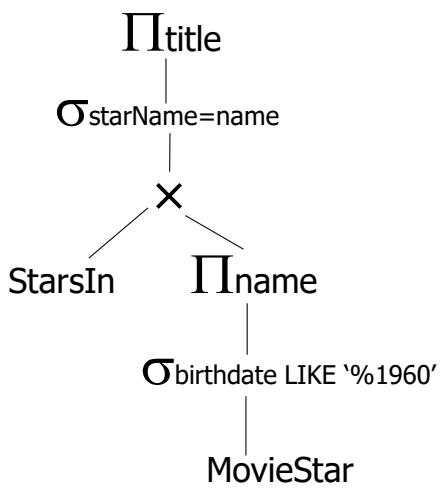
```
SELECT title
FROM StarsIn
WHERE starName IN (
        SELECT name
        FROM MovieStar
        WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

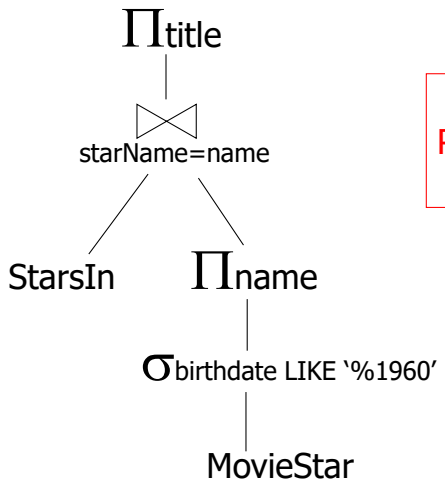
Example: Parse Tree



Example: Logical Query Plan

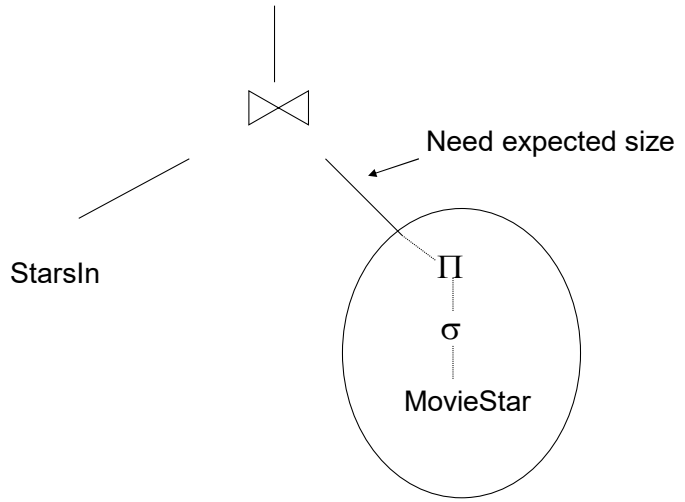


Example: Improved Locical Query Plan

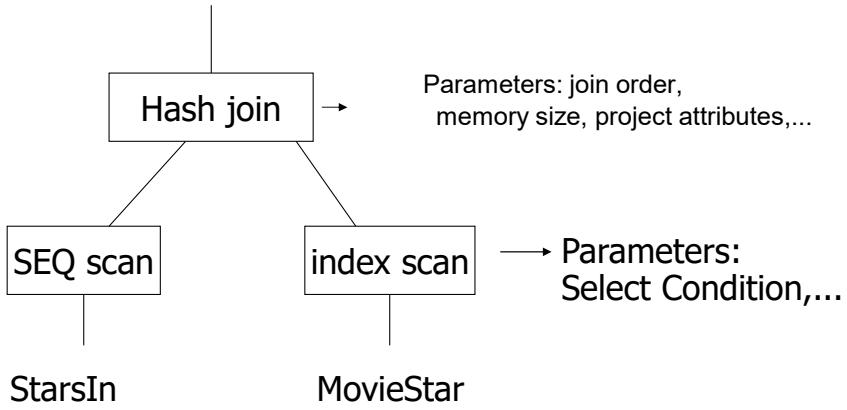


Question:
Push project to StarsIn?

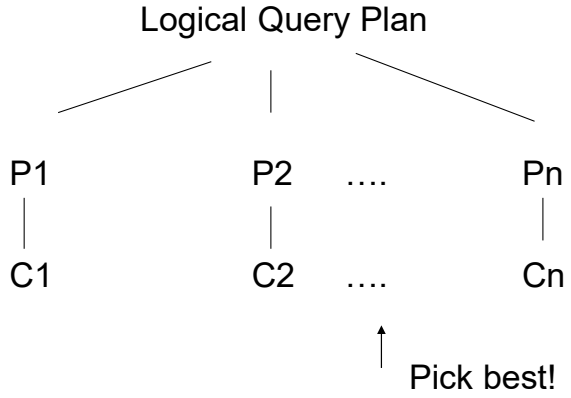
Example: Estimate Result Sizes



Example: One Physical Plan



Example: Estimate Costs



Algebraic Transformations

Generating Plans

- Start with query definition.
  - A plan, but usually a terrible one.
- Apply algebraic transformations to find other plans.
- Relational algebra is a good start, but we need also to consider: GROUP BY, duplicate elimination, HAVING, ORDER BY.

Algebraic Transformations

- Rules give equivalent expressions. meaning that whatever relations are substituted for variables, the results are the same.

Rules: Natural joins & cross products & union

$R \bowtie S = S \bowtie R$

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

$R \times S = S \times R$

$(R \times S) \times T = R \times (S \times T)$

$R \cup S = S \cup R$

$R \cup (S \cup T) = (R \cup S) \cup T$

But beware of thetajoin (join condition different from =)

- associative law does not hold.

Rules: Selects

$\sigma_{p1 \wedge p2}(R) = \sigma_{p1}[\sigma_{p2}(R)]$

$\sigma_{p1 \vee p2}(R) = [\sigma_{p1}(R)] \cup [\sigma_{p2}(R)]$

Rules: Project

Let: X = set of attributes
Y = set of attributes
XY = X U Y

$\pi_{xy}(R) = \pi_x[\pi_y(R)]$

Rules: σ + $\bowtie$ combined

Let p = predicate with only R attributes
q = predicate with only S attributes
m = predicate with only R,S attributes

$\sigma_p (R \bowtie S) = [\sigma_p (R)] \bowtie S$

$\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]$

Rules: σ + $\bowtie$ combined

Some rules can be derived:

$\sigma_{p \wedge q} (R \bowtie S) = [\sigma_p (R)] \bowtie [\sigma_q (S)]$

$\sigma_{p \wedge q \wedge m} (R \bowtie S) = \sigma_m [(\sigma_p R) \bowtie (\sigma_q S)]$

$\sigma_{p \vee q} (R \bowtie S) = [(\sigma_p R) \bowtie S] \cup [R \bowtie (\sigma_q S)]$

Rules: π, σ combined

Let x = subset of R attributes
z = attributes in predicate P
(subset of R attributes)

$\pi_x[\sigma_p (R)] = \pi_x \{\sigma_p [\pi_{xz} (R)]\}$

Rules: π, $\bowtie$ combined

Let x = subset of R attributes
y = subset of S attributes
z = intersection of R, S attributes

$\pi_{xy} (R \bowtie S) = \pi_{xy} \{[\pi_{xz} (R)] \bowtie [\pi_{yz} (S)]\}$

Rules: π, $\bowtie$, and σ combined

$$\pi_{xy}\{\sigma_p\ (R \bowtie S)\} =$$

$$\pi_{xy}\{\sigma_p\ [\pi_{xz'}(R) \bowtie \pi_{yz'}(S)]\}$$

$z' = z \cup \{\text{attributes used in } P\}$

Rules: σ, U combined

$\sigma_p(R \cup S) = \sigma_p(R) \cup \sigma_p(S)$

$\sigma_p(R - S) = \sigma_p(R) - S = \sigma_p(R) - \sigma_p(S)$

Which are "good" transformations?

$\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$

$\sigma_p(R \bowtie S) \rightarrow [\sigma_p(R)] \bowtie S$

$R \bowtie S \rightarrow S \bowtie R$

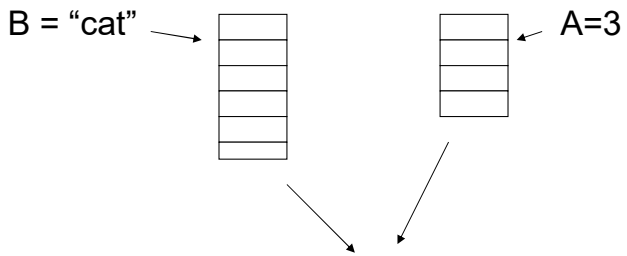
$\pi_x[\sigma_p\ (R)] \rightarrow \pi_x\{\sigma_p[\pi_{xz}\ (R)]\}$

Conventional wisdom: do projects early

Example: R(A,B,C,D,E) x={E}
P: (A=3) $\wedge$ (B="cat")

$\pi_x\{\sigma_p\ (R)\}$ vs. $\pi_E\{\sigma_p\{\pi_{ABE}(R)\}\}$

What if we have A, B indexes?



Intersect pointers to get
pointers to matching tuples

Bottom line

There is no transformation that is good in any case

Usually good: early selections

- Variant: move selection up to the root and then down on multiple path

Estimating the Cost of a Query Plan

Goal is to count disk I/O's.

But we first have to estimate sizes of intermediate results.

Keep statistics for relation R

- T(R) : # tuples in R
- S(R) : # of bytes in each R tuple
- B(R) : # of blocks to hold all R tuples
- V(R, A) : # distinct values in R for attribute A
- DOM (R, A): # possible distinct values for attribute A (size of domain for A)

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string
B: 4 byte integer
C: 8 byte date
D: 5 byte string

T(R) = 5
V(R,A) = 3
V(R,B) = 1

S(R) = 37
V(R,C) = 5
V(R,D) = 4

Size estimates for W = R1 x R2

$T(W) = T(R1) \times T(R2)$

$S(W) = S(R1) + S(R2)$

Size estimate for W = $\sigma_{A=a}$ (R)

$S(W) = S(R)$

$T(W) = ?$

Estimate Depends on Assumption

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3
V(R,B)=1
V(R,C)=5
V(R,D)=4

$W = \sigma_{z=val}(R) \quad T(W) = \frac{T(R)}{V(R,Z)}$

Assumption: Values in select expression Z = val
are uniformly distributed over possible V(R,Z) values.

Alternate Assumption

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

Alternate assumption
V(R,A)=3 DOM(R,A)=10
V(R,B)=1 DOM(R,B)=10
V(R,C)=5 DOM(R,C)=10
V(R,D)=4 DOM(R,D)=10

$W = \sigma_{z=val}(R) \quad T(W) = \frac{T(R)}{DOM(R,Z)}$

Assumption: Values in select expression Z = val
are uniformly distributed over possible DOM(R,Z) values.

Selections Involving Inequality

What about $W = \sigma_{z \geq val}(R)$?

T(W) = ?

Solution # 1: ~~T(W) = T(R)/2~~

- Assumption: All split values are equally likely

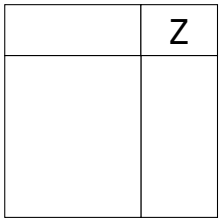
Solution # 2: T(W) = T(R)/3

- Assumption: Queries involving inequality ask more likely for a small fraction of possible tuples
- This assumption is usually preferred

Selections Involving Inequality (cont.)

Solution # 3: Estimate values in range

Example R



$W = \sigma_{z \geq 15}(R)$

$f = \frac{20-15+1}{20-1+1} = \frac{6}{20}$ (fraction of range)

$T(W) = f \times T(R)$

Size estimate for $W = R1 \bowtie R2$

Let x = attributes of R1
y = attributes of R2

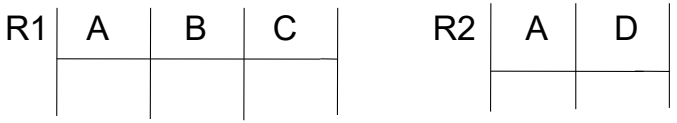
Case 1

$X \cap Y = \varnothing$

Same as R1 x R2

Size estimate for $W = R1 \bowtie R2$

Case 2 $W = R1 \bowtie R2$ $X \cap Y = A$

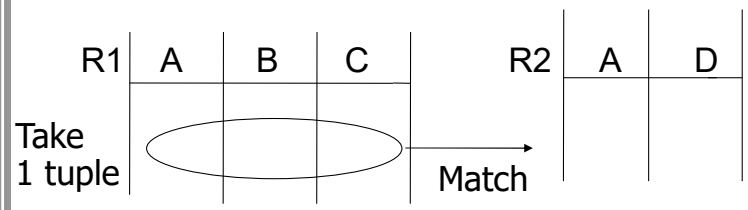


Assumption:

$V(R1,A) \leq V(R2,A) \Rightarrow$ Every A value in R1 is in R2
$V(R2,A) \leq V(R1,A) \Rightarrow$ Every A value in R2 is in R1

Size estimate for $W = R1 \bowtie R2$

Computing T(W) when $V(R1,A) \leq V(R2,A)$



1 tuple matches with $\frac{T(R2)}{V(R2,A)}$ tuples

so $T(W) = \frac{T(R2) \times T(R1)}{V(R2, A)}$

Implementation of DBMS

Size estimate for $W = R1 \bowtie R2$

$V(R1,A) \leq V(R2,A)$: $T(W) = \frac{T(R2)\,T(R1)}{V(R2,A)}$

$V(R2,A) \leq V(R1,A)$: $T(W) = \frac{T(R2)\,T(R1)}{V(R1,A)}$

In general:

$$T(W) = \frac{T(R2)\,T(R1)}{\max\{ V(R1,A), V(R2,A) \}}$$

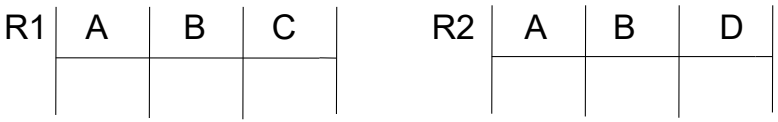
In all cases:

$S(W) = S(R1) + S(R2) - S(A)$

Implementation of DBMS

Size estimate for $W = R1 \bowtie R2$

Case 3 $W = R1 \bowtie R2$ $|X \cap Y| > 1$



We can generalize the approach: The product of T(R1) and T(R2) is divided by the maximum of V(R1, K) and V(R2, K) for each attribute K that is common to R1 and R2

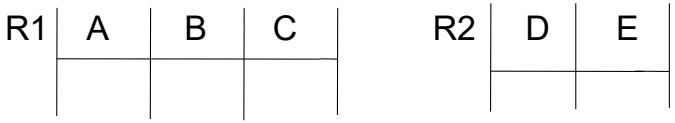
In the example above:

$$T(W) = \frac{T(R1)\,T(R2)}{\max\{ V(R1,A), V(R2,A) \}\ \max\{ V(R1,B), V(R2,B) \}}$$

Implementation of DBMS

Size estimate for Equijoins

$W = R1 \underset{A=D}{\bowtie} R2$



The number of tuples can be calculated in a similar way as for a natural join.

$$T(W) = \frac{T(R1)\,T(R2)}{\max\{ V(R1,A), V(R2,D) \}}$$

Note, that the size of a tuple is different for an equijoin:

$S(W) = S(R1) + S(R2)$

For Complex Expressions We Need Intermediate Values for T,S,V

E.g. $W = [\sigma_{A=a}(R1)] \bowtie R2$

Treat as relation U

$T(U) = T(R1)/V(R1,A)$

$S(U) = S(R1)$

Problem: We also need V (U, *)

Example

R1

A	B	C	D
cat	1	10	10
cat	1	20	20
dog	1	30	10
dog	1	40	30
bat	1	50	10

$V(R1,A)=3$
$V(R1,B)=1$
$V(R1,C)=5$
$V(R1,D)=3$

$U = \sigma_{A=a}(R1)$

$V(U,A)=1 \quad V(U,B)=1 \quad V(U,C) = \frac{T(R1)}{V(R1,A)}$

V(U, D) somewhere in between

Estimates for Selections

$U = \sigma_{A=a}(R)$

$V(U,A) = 1$

$V(U,X) = V(R,X)$ for $x \neq A$

“preservation of value sets”

Estimates for Joins

$U = R1(A,B) \bowtie R2(A,C)$

$V(U,A) = \min \{ V(R1, A), V(R2, A) \}$

$V(U,B) = V(R1, B)$

$V(U,C) = V(R2, C)$

also “preservation of value sets”

Example

$Z = R1(A,B) \bowtie R2(B,C) \bowtie R3(C,D)$

R1	T(R1) = 1000	V(R1,A)=50	V(R1,B)=100
R2	T(R2) = 2000	V(R2,B)=200	V(R2,C)=300
R3	T(R3) = 3000	V(R3,C)=90	V(R3,D)=500

Example

Partial Result: $U = R1 \bowtie R2$

$$T(U) = \frac{1000 \times 2000}{200}$$

V(U,A) = 50
V(U,B) = 100
V(U,C) = 300

$Z = U \bowtie R3$

$$T(Z) = \frac{1000 \times 2000 \times 3000}{200 \times 300}$$

V(Z,A) = 50
V(Z,B) = 100
V(Z,C) = 90
V(Z,D) = 500

Summary

Estimating size of results is an “art”

Prerequisite for estimates are statistics about the individual relations

- Statistics must be kept up to date

Join Algorithms

Comparing Join Algorithms

Options:

Transformations: R1 ⋈ R2, R2 ⋈ R1

- Join algorithms:
  - Iteration (nested loops join)
  - Merge join
  - Join with index
  - Hash join

Factors that affect performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Running Example

Example: R1 ⋈ R2 over common attribute C

T(R1) = 10,000

T(R2) = 5,000

S(R1) = S(R2) = 1/10 block

Memory available = 101 blocks

→ Metric: # of IOs (ignoring writing of result)

Each block holds 10 tuples → R1=1000 blocks, R2=500 block
(from it: 10,000 tuples, 10 tuples per block → 1,000 blocks) ....

Iteration Join (Nested Loops Join)

```
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then output r,s pair
```


Example: Iteration Join R1 ⋈ R2

Relations not contiguous

Recall
- T(R1) = 10,000 T(R2) = 5,000
- S(R1) = S(R2) =1/10 block
- MEM=101 blocks

Cost: for each R1 tuple:

[Read tuple + Read R2]

Total =10,000 [1+5000] = 50,010,000 IOs

Can we do better?

Use our memory

(1) Read tuples from R1 into 100 main memory blocks

(2) Read all of R2 (using 1 block) + join

(3) Repeat until done

Example: Improved Iteration Join

If 100 blocks are used to read R1 into memory:
number of tuples per chunk: 100*10 = 1000 tuples.....

Cost: for each R1 chunk:

Read chunk: 1000 IOs
Read R2: 5000 IOs
6000 IOs

$$\text{Total} = \frac{10{,}000}{1{,}000} \times 6000 = 60{,}000 \text{ IOs}$$

the read chunk = 1000 is derived from:
[redacted] IOs per chunck.....

Can we do better?

☛ Reverse join order: R2 ⋈ R1

Total = $\frac{5000}{1000}$ x (1000 + 10,000) =

5 x 11,000 = 55,000 IOs

(from earlier: 10,000 tuples, 10 tuples per block → 1,000 block

Modified Example:

Relations contiguous

Cost

For each R2 chunk:

Read chunk: 100 IOs

Read R1: 1000 IOs

1,100

Total= 5 chunks x 1,100 = 5,500 IOs

others for each R1 chunk: result 6000 I/Os ....

Merge Join

Merge join (conceptually)

(1) if R1 and R2 not sorted, sort them

(2) i ← 1; j ← 1;

While (i ≤ T(R1)) ∧ (j ≤ T(R2)) do

if R1{ i }.C = R2{ j }.C then outputTuples

else if R1{ i }.C > R2{ j }.C then j ← j+1

else if R1{ i }.C < R2{ j }.C then i ← i+1

Merge Join (cont.)

Procedure Output-Tuples

While (R1{ i }.C = R2{ j }.C) ∧ (i ≤ T(R1)) do

[jj ← j;

while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do

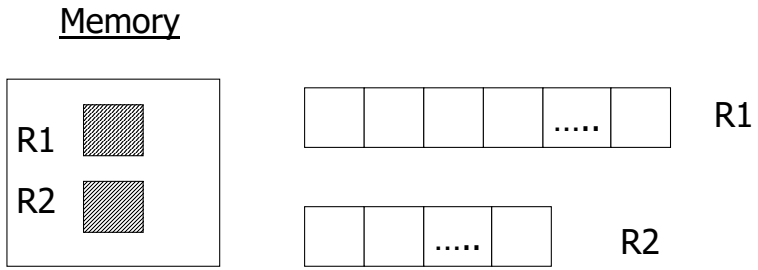
[output pair R1{ i }, R2{ jj };

jj ← jj+1]

i ← i+1]

Example: Merge Join

Assumption: Both R1, R2 ordered by C; relations contiguous



Total cost: Read R1 cost + read R2 cost

= 1000 + 500 = 1,500 IOs

Modified Example: Merge Join

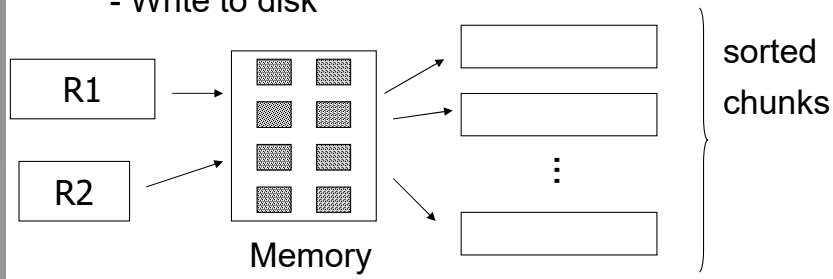
R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first…. HOW?

Recall: 2PMMS

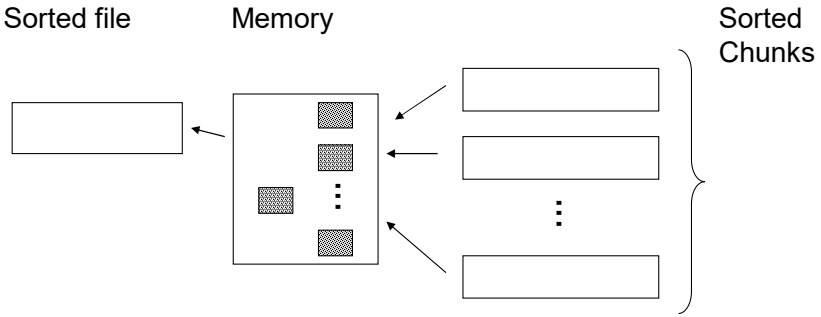
(i) For each 100 block chunk of R:

- Read chunk
- Sort in memory
- Write to disk



2PMMS (cont.)

(ii) Read all chunks + merge + write out



Cost 2PMMS

Each tuple is read,written, read, written

Thus,

Sort cost R1: 4 x 1,000 = 4,000

Sort cost R2: 4 x 500 = 2,000

Modified Example: Merge Join (cont.)

R1,R2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

But: Iteration join cost = 5,500
so merge joint does not pay off!

Merge Join vs. Iteration Join

Iteration join is essetially quadratic whereas merge join is linear

Thus, the decision depends on the size of the relations:

Example: R1 = 10,000 blocks both contiguous
R2 = 5,000 blocks and not ordered

Iterate: $\frac{5000}{100}$ x (100+10,000) = 50 x 10,100
= 505,000 IOs

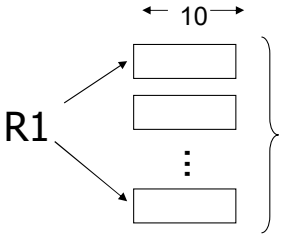
Merge join: 5 (10,000+5,000) = 75,000 IOs

Merge Join (with sort) Wins!

How much memory do we need for merge sort?

Example:

- contiguous relation with 1000 Blocks
- 10 memory blocks



100 chunks ⇒ to merge, need
100 input blocks!

In general:

Say M blocks in memory

B blocks for the relation to be sorted

chunks = $\lceil (B/M) \rceil$ size of chunk = M

chunks < buffers available for merge

Thus $\lceil (B/M) \rceil < M$

or approximately $M^2 > B$ or $M > \lceil \sqrt{B} \rceil$

In our example

R1 is 1000 blocks, $M > \lceil 31.62 \rceil$

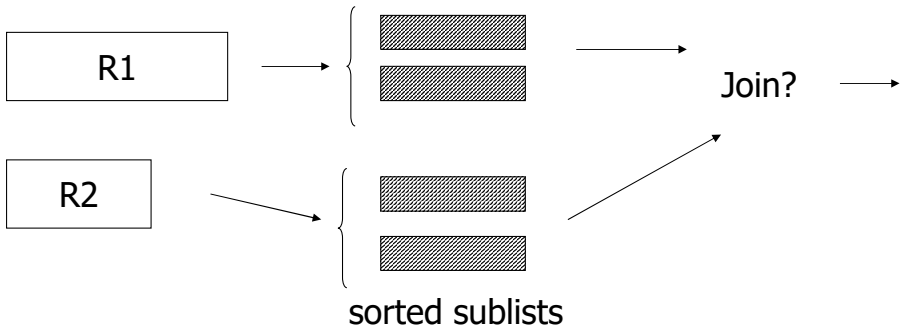
R2 is 500 blocks, $M > \lceil 22.36 \rceil$

Therefore, we need at least 33 buffer blocks in main memory

Can we improve on merge join?

Idea: do we really need the fully sorted files?

No! With enough main memory we can combine the last phase of the sorting algorithm with the actual join



Cost of improved merge join:

C = Read R1 + write R1 into sorted sublists

+ read R2 + write R2 into sorted sublists

+ join

= 2000 + 1000 + 1500 = 4500

Limitations: more memory required during join:

- In general we require $M^2 > B(R1) + B(R2)$
- For our example R1 and R2 we sort R1 (using 101 buffers) into 10 sorted sublists and R2 into 5 sublists.
- In the merge phase, we need at least 15 buffers, one per sublist, for input.
- That leaves additional 86 buffer blocks for records that share a C-value.

Hash Join

Hash join (conceptual)

- Hash function h, range 0 → k
- Buckets for R1: G0, G1, ... Gk
- Buckets for R2: H0, H1, ... Hk

Algorithm

(1) Hash R1 tuples into G buckets

(2) Hash R2 tuples into H buckets

(3) For i = 0 to k do

match tuples in Gi, Hi buckets

Simple example hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

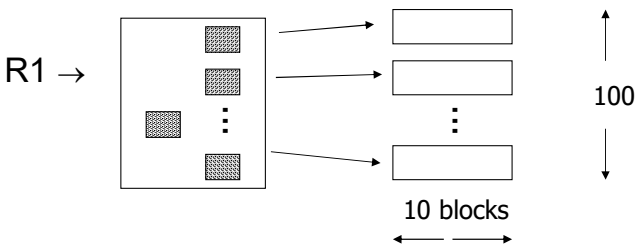
	R1	R2
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

Example Hash Join

R1, R2 contiguous (un-ordered)

→ Use 100 buckets

→ Read R1, hash, + write buckets

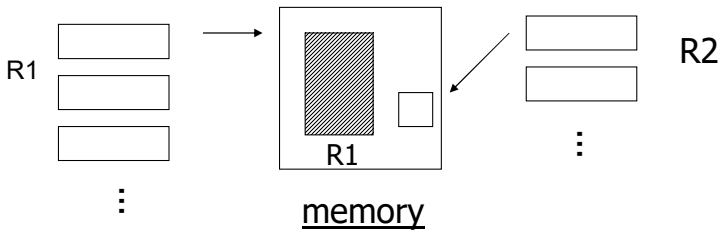


Example Hash Join

-> Same for R2

-> Read one R1 bucket; build memory hash table

-> Read corresponding R2 bucket + join



Then repeat for all buckets

Cost

"Bucketize:" Read R1 + write

Read R2 + write

Join: Read R1, R2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum memory requirements:

Bucketizing: The number of buckets can be at most M-1

Assuming that all buckets have roughly the same size, the size of the bucket is = B / (M-1)

M = number of memory buffers

B = number of R blocks

Joining of individual buckets: The bucket of one relation has to be completely in main memory (in detail: fit in M-1 blocks)
Therefore: B/(M-1) <= M-1

or approximately M > $\lceil\sqrt{B}\rceil$

Note: In contrast to Merge join it is sufficient if this relationship holds for the smaller relation

Sort-based vs Hash-Techniques

The Hash-based algorithm has a size requirement that depends only on the smaller of the two relations rather than the sum of the argument sizes as for the optimized sort-based algorithm

The Sort-based algorithm allows us to produce a result in a sorted order and take advantage of that later

- The result might be used in another sort-based algorithm later, or
- It could be the answer to a query that is required to be produced in sorted order

The Hash-based algorithm depends on the buckets being of equal size. However, there is generally a variation in size.
Result:

- It is not possible to use buckets that on average need the whole available main-memory. We must limit them to a smaller figure
- This affect is especially prominent, if the number of different hash keys is small

Different Number of Passes (1)

The execution of some algorithms relies on the availability of a certain number of memory buffers

- sort-based algorithms
- hash-based algorithms

Remind:

with

M = number of memory buffers

B = number of blocks for the relation(s)

we have to have approximately $M^2 > B$

The algorithms use two passes

- one pass to prepare the data (sorted sublists, buckets)
- a second pass to perform the desired action (e.g., join)

Different Number of Passes (2)

If we have relations of a larger size we can add one or more additional pass

- sort-based algorithms: the subsequences are merged to produce a smaller number of larger subsequences
- hash-based algorithms: each bucked is further divided into smaller buckets using a second hash function

Performance

- each additional pass requires reading and writing the relation

Size of the relation

- each additional pass increases the allowed size of the relation by a factor M
- using n passes, we have $M^n > B$

Different Number of Passes (3)

The other extreme: we have enough memory to accommodate one relation completely in main memory, i.e.

M > B

In this case we can omit one pass and get a trivial (one-pass) join-algorithm:

- load one relation completely in main-memory
- read the second relation one block at a time and join the tuples with the first relation in main-memory

Selection

Recall: Notions of Clustering

Clustered-file organization

- tuples of one relation R are stored in blocks together with tuples of some other relation S with which they share a common value
  - to optimize the join of the two relations

Clustered relation (= contiguous storage)

- tuples of the relation are stored in blocks that are exclusively or at least predominantly devoted to storing that relation

Clustering index

- an index in which the tuples having a given value of the search key appear in blocks that are largely devoted to storing tuples with that search-key value

Selection (1)

Key decision: shall we use an index and when we have the choice which one?

Task: Implementation of $\sigma_C(R)$, Metric: Disk I/Os

Options:

- Scan the complete relation
  - B(R) if R is clustered
  - T(R) if R is not clustered
- Condition C is an equality term such as a = 10, the a-value we search is uniformly distributed over V(R, a) and we use an index on attribute a
  - B(R) / V(R, a) if the index is clustering
  - T(R) / V(R, a) if the index is not clustering
- Condition C is an inequality term such as b < 20 and we use an index on attribute b
  - B(R) / 3 if the index is clustering
  - T(R) / 3 if the index is not clustering

Selection (2)

In case of index usage, we also have to account for disk I/O's to read some index blocks.

If we have several indexes available chose the one that produces the better result

- load tuples based on this index and check other conditions in main memory

Selection: Example (1)

Selection: $\sigma_{X=1 \text{ AND } Y=2 \text{ AND } Z<5}$ (R)

T(R) = 5000

B(R) = 200

V(R, x) = 100

V(R, y) = 500

R is clustered

There are indexes on each of x, y, and z but only the index of z is clustering

Selection: Example (2)

Scan relation: cost is B(R) = 200 I/Os

Use index on x to find those tuples with x=1 and check for each tuple the rest of the condition:
cost is T(R) / V(R, x) = 50 I/Os

Use index on y to find those tuples with y=2 and check for each tuple the rest of the condition:
cost is T(R) / V(R, y) = 10 I/Os

Use index on z to find those tuples with z<5 and check for each tuple the rest of the condition:
cost is B(R) / 3 = 66 + 2/3 I/Os