Physical Models (Indexing)

Query Optimization

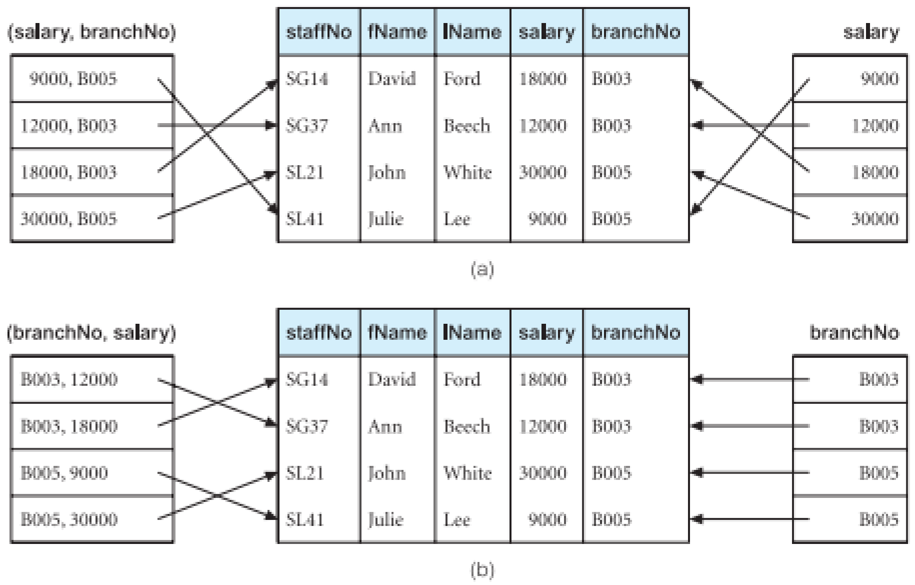
CSC 321/621 – 4/17/2012

Indexing

- Three primary types of indices:
  - Primary index (one of these per file/table)
    - Actual records stored in sorted order based on a key (unique) for the table; index based on the key as well
  - Clustering index (one of these per file/table, if primary not used – *why?*)
    - Actual records stored in sorted order based on a non- key for the table; index based on the non-key (meaning 1+ records with same value in index, pointing to different places)
  - Secondary index (multiple of these per file/table)
    - An index that is based an attribute that the data is not sorted on
- Note the indices themselves are always ordered (the data referenced doesn't have to be for secondary)

Indexing

- Index can be on composite attributes
  - Have an internal ordering (assume left-to-right based subfield ordering)
- A few example indices for the DreamHome Staff table



staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

Which of these appear to be: Primary indices?

Clustering indices?

Secondary indices?

Indexing

- Our goal for indexing was to effectively get rid of having to do sequential scan of data file (tables)

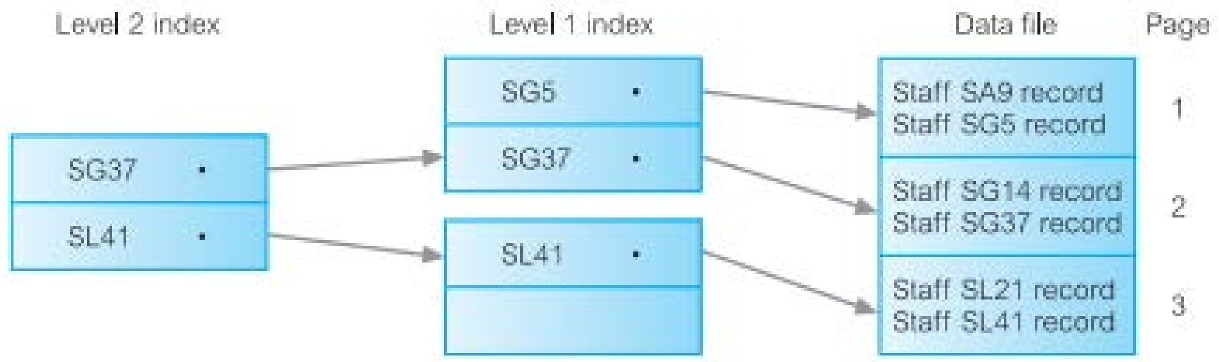
- Aren't we doing now a linear search, just on a different file?
  - *Yes, though remember we can stop early!*

Indexing

- What if we introduced an index to an index?
- Need a new idea first: A sparse index
- A *dense* index:
  - One index entry for each actual entry
- A *sparse* index:
  - Does not contain an index for every actual record
  - Instead, gets you to a “close-by” region of the file
- Our 2nd index needs to be a sparse index

Indexing

- Note that both indices in this multi-level index example are sparse
  - They only show the upper bound value for the pages they point to.
  - You are linear searching through parts of multiple indices



B-Trees

- Probably the single most popular index implementation today is the idea of B-Trees, which are an implementation of this index-of-an-index idea
  - B is for “balanced”

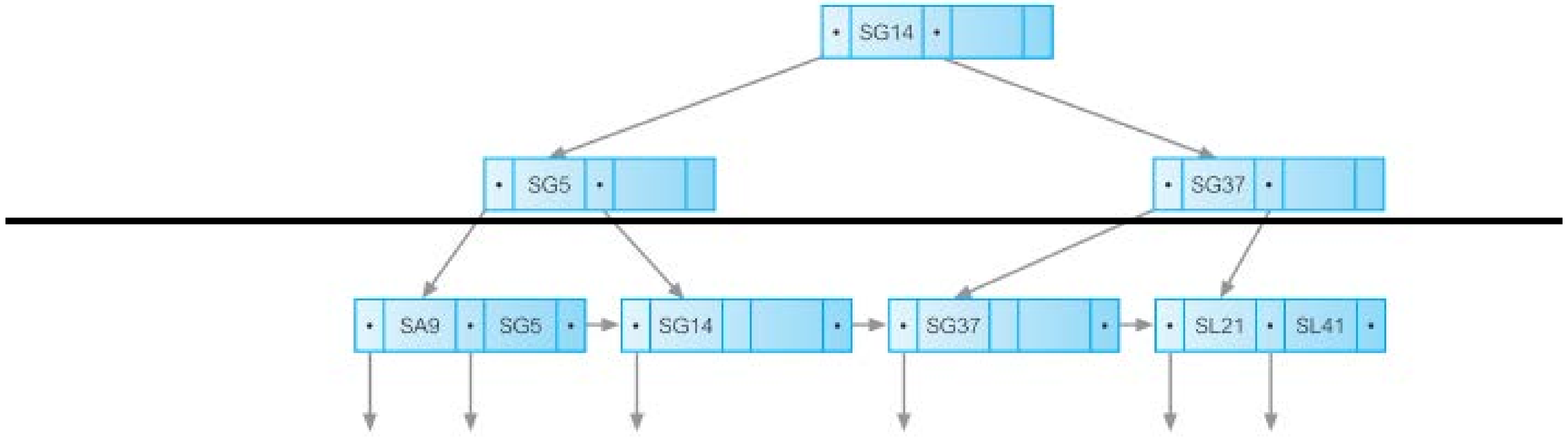


Figure F.11 Example of B$^+$-tree index.

B^+-Trees: Definition

- If the root is not a leaf node, it must have at least two children
- The number of key values in a non leaf node is 1 less than the number of pointers to children
- For a tree of order n (n = max degree per node)
  - Each node (except root and leaves) must have between n/2 and n pointers and children.
    - Round up if n/2 is not an integer
  - By second rule above, then the number of key values in a leaf node must be between (n-1)/2 and n-1.
- The tree must always be balanced
  - Every path from root to leaf must have same length
- Leaf nodes are linked in order of key values
  - Links are pointers to the next leaf

B-Trees

- Is this example tree balanced?
- Are the leaf nodes linked in order of increasing key values?
- Are number of keys in a node 1-less than number of pointers (barring leaf nodes)?

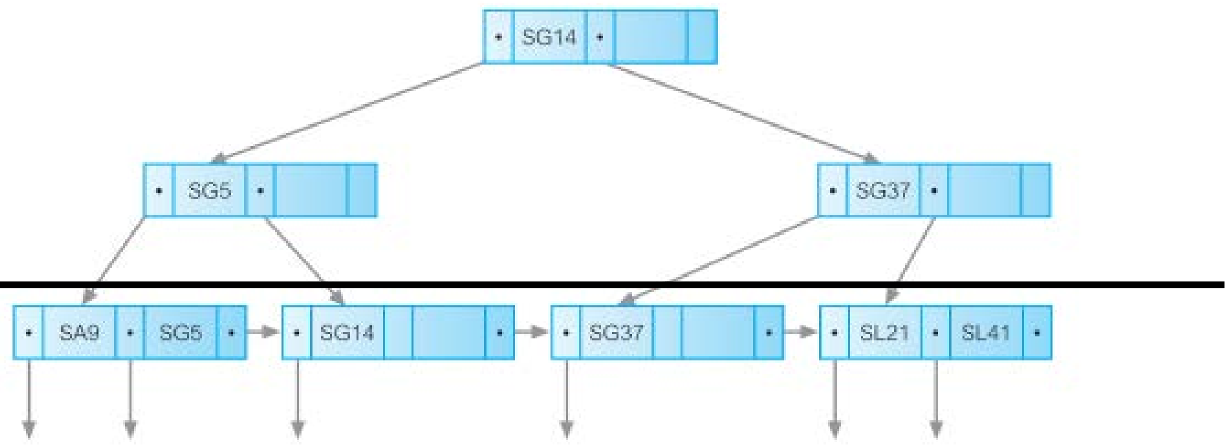


Figure F.11 Example of B$^+$-tree index.

B+-Trees: Search Process

- To have low access times, try to keep the depth (root → leaf path distance) low.

- Starting at root, scan left-to-right through keys
- If value of interest is less than or equal to current key, follow left pointer
- Otherwise move one key to right

B+-Trees: Searching

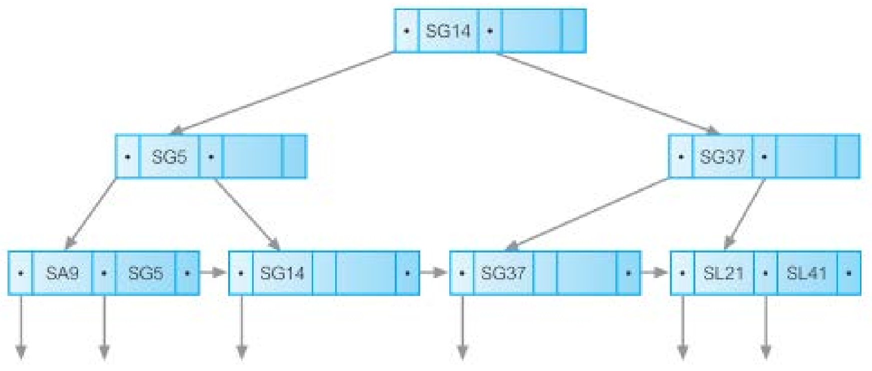


Figure F.11 Example of B+-tree index.

- Searching for SL21:
  - Root node, current key = SG14
    - SL21 > SG14, so move over a key
      - Nonexistent next key, so take right pointer
  - SG37 node, current key = SG37
    - SL21 > SG37, so move over a key
      - Nonexistent next key, so take right pointer
  - SL21 node, current key = SL21
    - SL21 == SL21, so take left pointer to actual value

B+-Trees: How Much Data?

- How much data can be represented in a 3-level B+-tree?
  - Typically, a node is an entire disk page, which are commonly 4k (4096) bytes
  - Book suggests that each node has “next pointer” to next node on same level
  - Given an integer key (an assumption), each key in a node requires 4 bytes
  - Each child pointer in a node requires 4 bytes

B+-Trees: How Much Data?

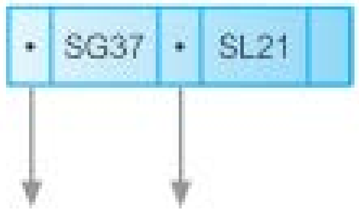
- Level 1: 4096-4 / (4+4) → 511 keys in root node (+ 4 more bytes left for another child pointer), leading to 512 children
- Level 2: Each child can have 511 keys and 512 children, leading to 512 * 511 = 261632 more keys and 512*512 =262144 pointers
- Level 3: Each of the children referenced by the 262144 pointers can have 511 keys, leading to 133,955,584 keys
- Grand total: 134,217,727 values can be maintained in a standard page-oriented index
- Takes at most 4 steps to access any of those (level1(root) → level2 → level3 → actual data)

B+-Tree: Maintenance Procedure

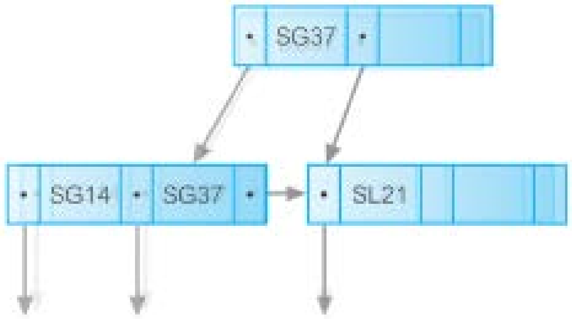
- Insertion:
  - Search to determine where item should go
    - A bucket on the last (leaf) level
  - If bucket is not full, add item to it
  - Else
    - Allocate new leaf and move half of discovered bucket's elements to new leaf/bucket
    - If the parent is not full, insert largest component of current leaf into the parent at appropriate place
    - If the parent is full:
      - Split the parent
      - Add the middle key of the current bucket to the parent node
      - Repeat until a parent is found that need not split
  - If root splits, create a new root with one key, two pointer

Maintaining a B+-Tree

- Example: assume degree 3 nodes
  - Initial tree:

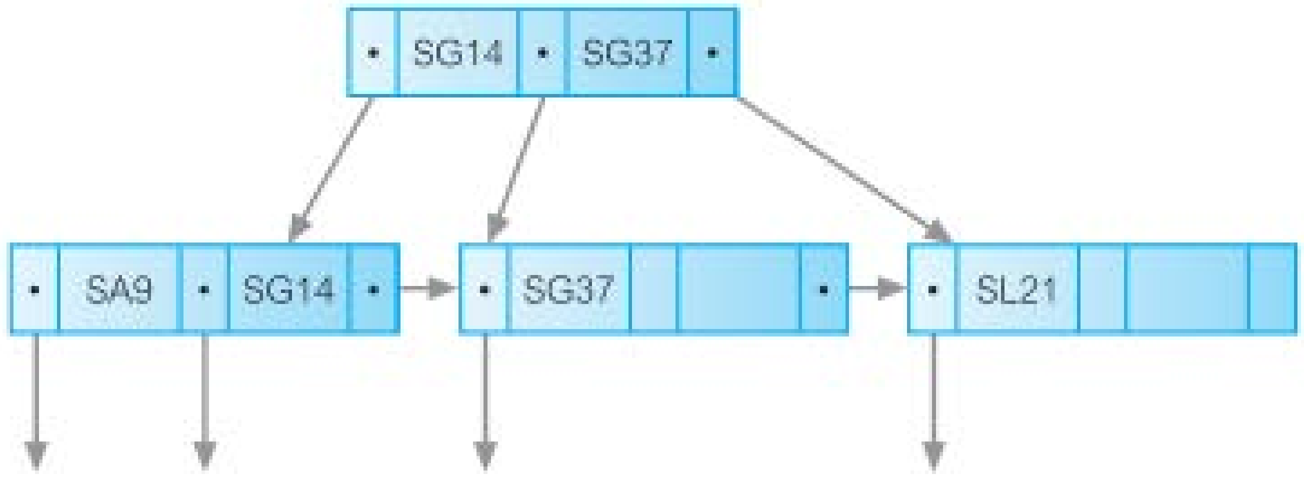


  - Inserting SG14, which should go before SG37
    - Causes new node to right, housing SL21
    - Causes new node above, with SG37 copy



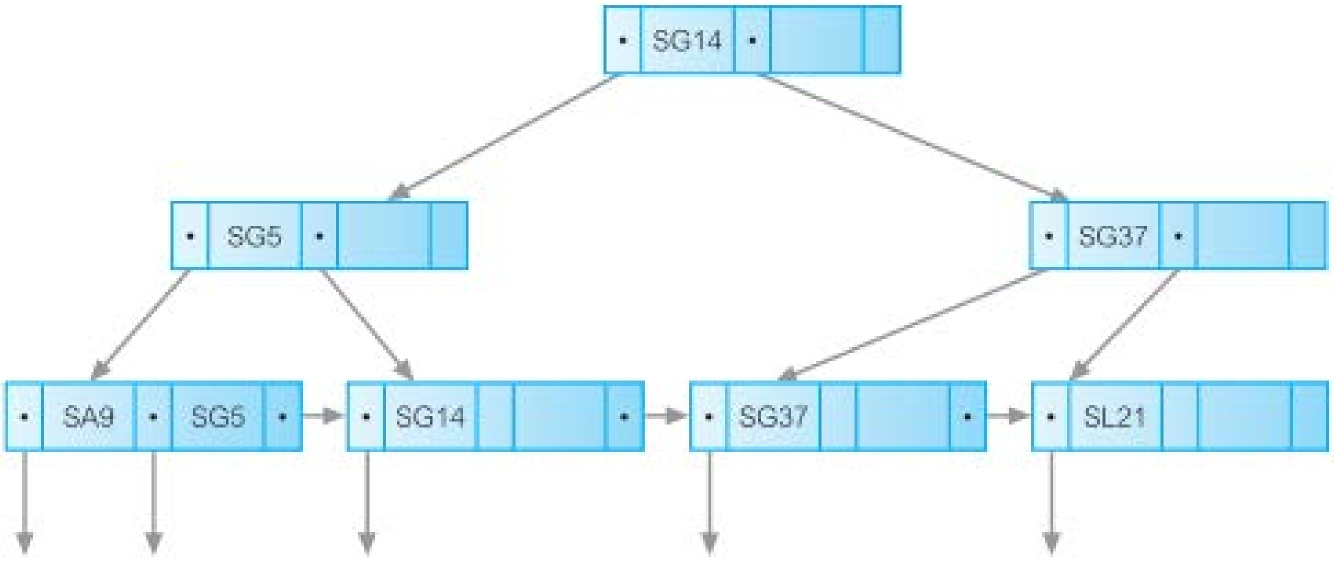
Maintaining a B+-Tree

– Inserting SA9, which should go before SA9
  - Causes new node to right, housing SG37
  - Causes copy to parent node, with SG14



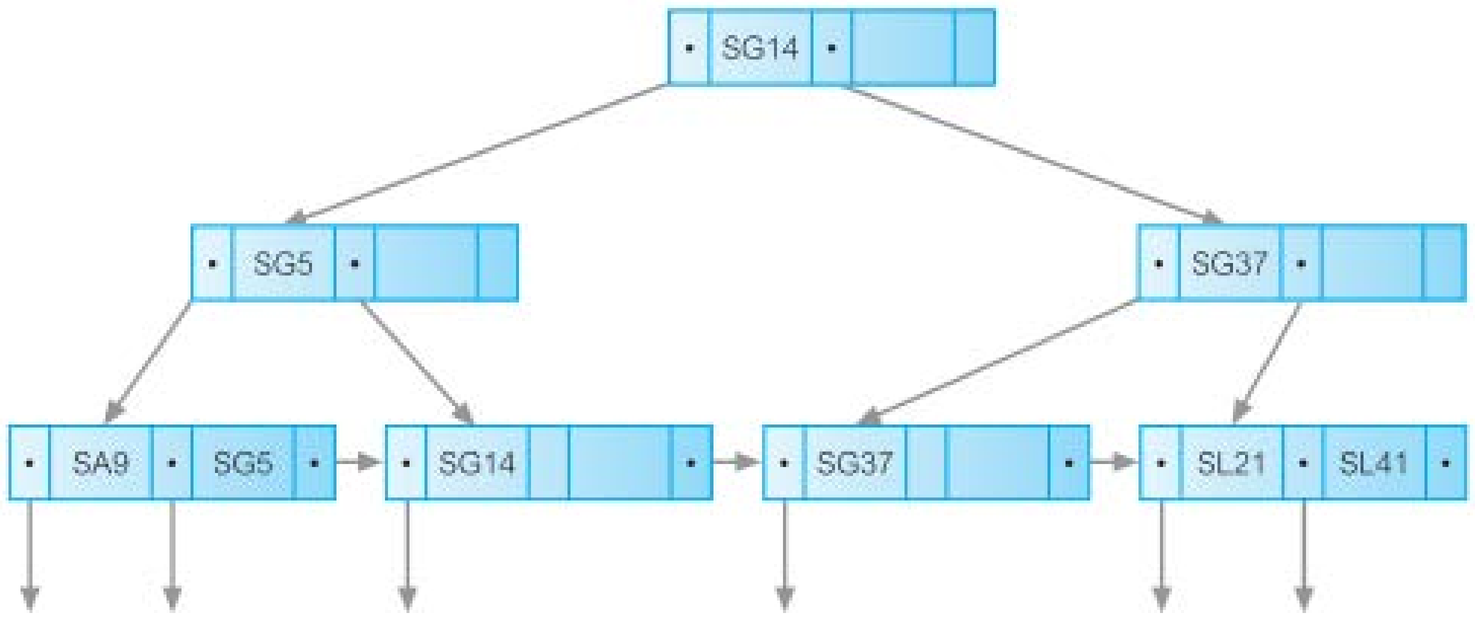
Maintaining a B+-Tree

- Inserting SG5, which should go between SA9 and SG14
  - Causes new node to right, housing SG14
  - Causes copy to parent node, with SG5, which fills parent node
  - That causes copy to a new parent node, with SG14



Maintaining a B+-Tree

– Inserting SL41, falls into last spot in last node



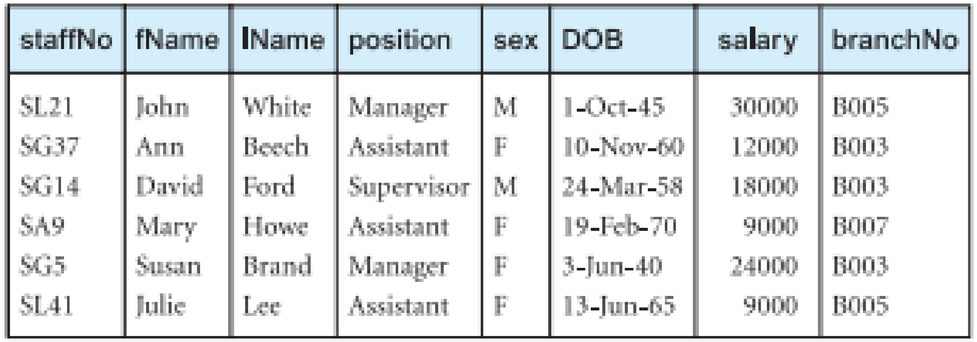
Indexing Techniques

- While B+-trees are popular, they are not the only indexing technique.

- Bitmap indices
  - Useful for sparse domains
- Join indices
  - Useful for speeding up common queries requiring 2+ tables (joins)

Bitmap Index

- For attributes that have a small range (such as an enumeration of values), maintain a tuple X domain-values bit matrix
- Place a 1 in each entry where a tuple has the represented domain-value, 0 otherwise

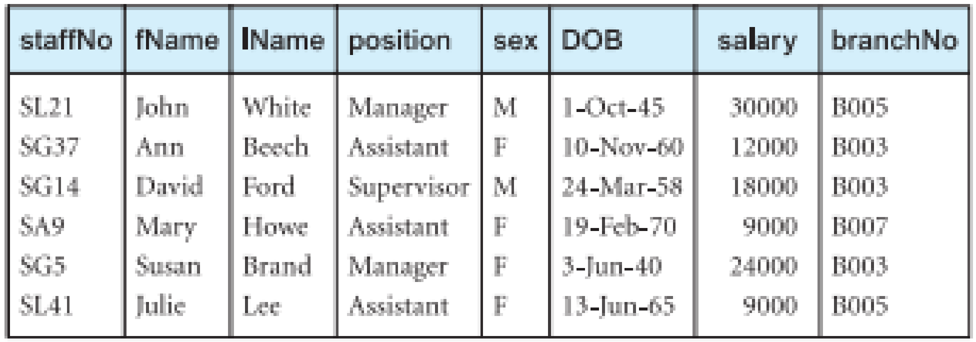


staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Assume only three positions:
Manager, Assistant, Supervisor

Assume only three branches:
B003, B005, B007

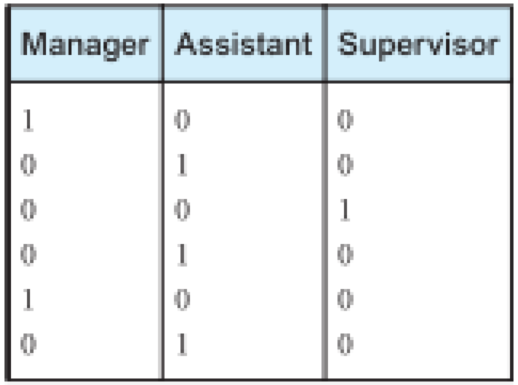
Bitmap Index



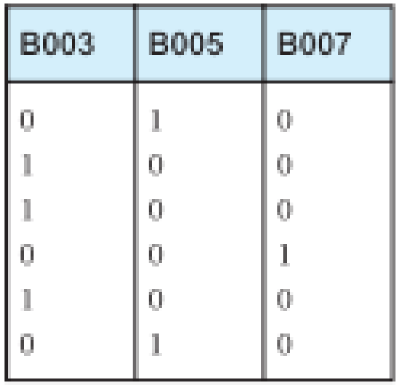
staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Assume only three positions:
Manager, Assistant, Supervisor

Assume only three branches:
B003, B005, B007



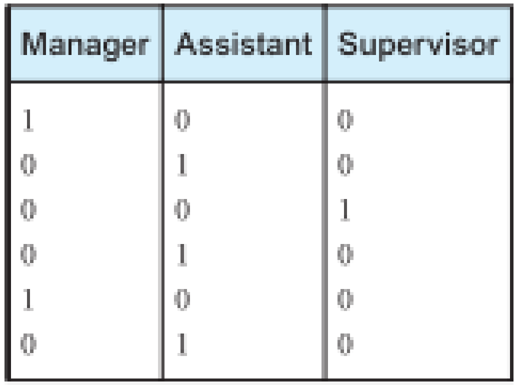
Manager	Assistant	Supervisor
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0
0	1	0



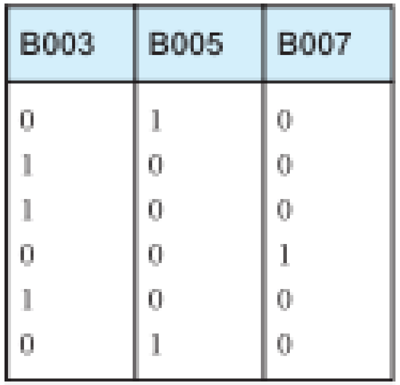
B003	B005	B007
0	1	0
1	0	0
1	0	0
0	0	1
1	0	0
0	1	0

Bitmap Index

- Advantages of bitmap indices:
  - Compact representations (bitwise if needed)
  - Queries on represented attributes can exploit boolean logic operations
    - SELECT * FROM Employee WHERE position=‘Manager’ and branch=‘B003’



Manager	Assistant	Supervisor
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0
0	1	0

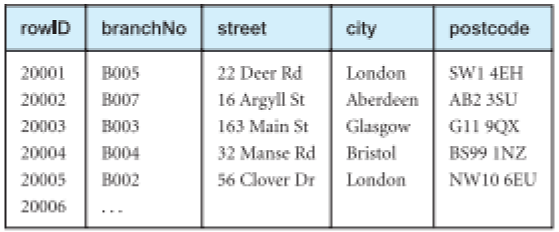


B003	B005	B007
0	1	0
1	0	0
1	0	0
0	0	1
1	0	0
0	1	0

Join Indices

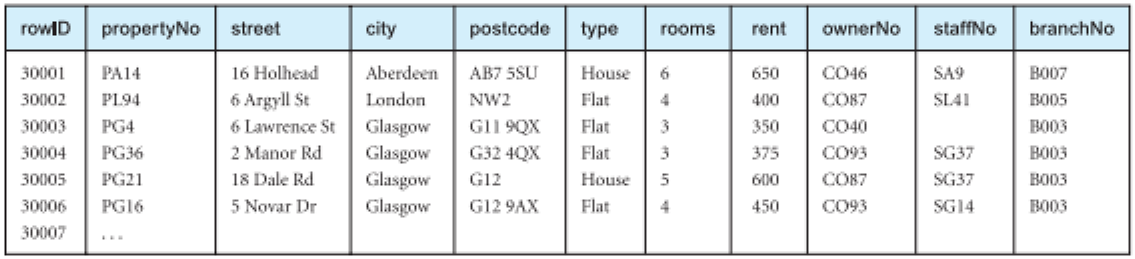
- A join index is an index built off a shared-domain attribute of 2 or more tables

Branch



rowID	branchNo	street	city	postcode
20001	B005	22 Deer Rd	London	SW1 4EH
20002	B007	16 Argyll St	Aberdeen	AB2 3SU
20003	B003	163 Main St	Glasgow	G11 9QX
20004	B004	32 Manse Rd	Bristol	BS99 1NZ
20005	B002	56 Clover Dr	London	NW10 6EU
20006	. . .			

PropertyForRent



rowID	propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
30001	PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
30002	PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
30003	PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
30004	PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
30005	PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
30006	PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003
30007	. . .									

- Potential/reasonable targets are city and branchNo
  - Shared by both relations

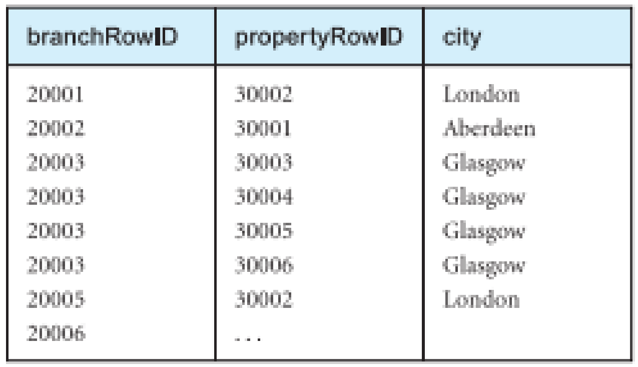
  What would be the likely purpose of our join-query if use 'city'?

  What would be the likely purpose of our join-query if use 'branch'?

Join Indices

- Join index contains primary keys of the relations, plus the join attribute
- Sorted by attribute of interest
  - Here, branch PK

Join Index



branchRowID	propertyRowID	city
20001	30002	London
20002	30001	Aberdeen
20003	30003	Glasgow
20003	30004	Glasgow
20003	30005	Glasgow
20003	30006	Glasgow
20005	30002	London
20006	. . .	

  - Joins typically quite expensive; allows you to pre-compute
    - *Is there a cost-at-insert-time here, like for B+-trees?*

MySQL for Indices

MySQL, for InnoDB tables, only supports BTREE indices

CREATE INDEX indexName ON tbl_name
(index_col_name,...)

Optimization

- Using indices and appropriate physical storage mechanisms can affect query processing times
- At a higher level, one can also consider query optimization – determining the best order for performing the subsets of a complex query

Approaches to Query Optimization

- Will examine heuristics (general approaches/good ideas)
- If have time, will look at cost analysis and optimization
  - Need statistics to be used effectively

- General goal (should sound familiar):
  - Reduce disk accesses
  - Reduce tuple touches

A Motivating Example

“Find all managers who work at the London Branches”

SELECT * FROM Staff s, Branch b

WHERE s.branchNo = b.branchNo AND (s.position=‘Manager’ AND b.city=‘London’)

Motivating Example

SELECT * FROM Staff s, Branch b

WHERE s.branchNo = b.branchNo AND (s.position='Manager' AND b.city='London')

Equivalent relational algebra expressions:

(1) $\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'}) \wedge (\text{Staff.branchNo}=\text{Branch.branchNo})}(\text{Staff} \times \text{Branch})$

(2) $\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'})}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch})$

(3) $(\sigma_{\text{position}=\text{'Manager'}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\sigma_{\text{city}=\text{'London'}}(\text{Branch}))$

Do you agree these are equivalent to the SQL?

Motivating Example

- Cost model (simplistic):
  - No indices
  - Results of intermediate operations stored on disk
  - Tuples are read one at a time
  - Final write cost is ignored (all queries have the same cost here, because get to same final answer)

Motivating Example

(1) $\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'}) \wedge (\text{Staff.branchNo}=\text{Branch.branchNo})}(\text{Staff} \times \text{Branch})$

(2) $\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'})}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch})$

(3) $(\sigma_{\text{position}=\text{'Manager'}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\sigma_{\text{city}=\text{'London'}}(\text{Branch}))$

Assume the following about the data size

Staff: 1000 tuples

Branch: 50 tuples

Managers: 50 (one for each branch)

Branches in London: 5

Motivating Example

$$\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'}) \wedge (\text{Staff.branchNo}=\text{Branch.branchNo})}(\text{Staff} \times \text{Branch})$$

Approach: Create CP of Staff, Branch, then apply predicate

Costs:

  1000 reads to load Staff

  50 reads to load Branch

  Temporarily generate 50,000 relation intermediate table

  Read all 50,000 entries and apply predicate

101,050 accesses

Motivating Example

$$\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'})}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch})$$

Use a join of Staff and Branch based on a condition, then sub-select using predicate

Assume this style join never generates the full intermediate CP

1000 reads to load Staff

50 reads to load Branch

Intermediate result of join has 1000 entries (each staff member associated with their branch information)

Process that 1000 entry table using selection

3,050 accesses

Motivating Example

$$(\sigma_{\text{position}=\text{'Manager'}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\sigma_{\text{city}=\text{'London'}}(\text{Branch}))$$

Sub-select from Staff and Branch independently, then join using a condition

? accesses

Motivating Example

$$(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$$

Sub-select from Staff and Branch independently, then join using a condition

50 reads for Branch table
5 writes of London branches into temporary table

1000 reads for Staff table
50 writes of Managers into temporary table

5 reads of London branches from temp table join
50 reads of Managers from temp table for join

1160 disk accesses

Motivating Example

Suggestion 1: We should attempt to do projections and selections ____________________.

As early as possible

Query Optimization

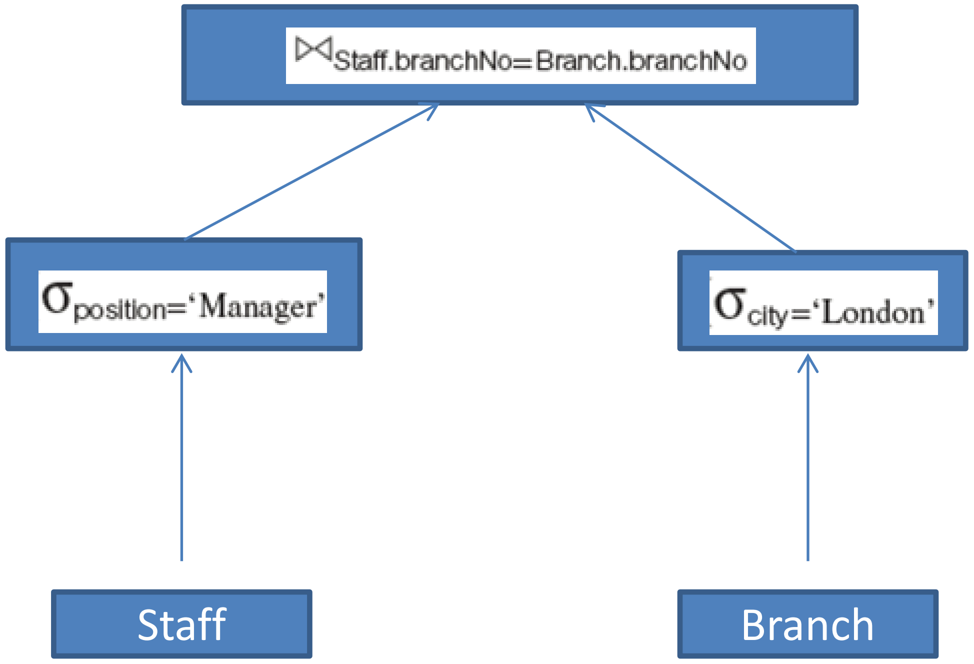
- When a query is parsed, we will assume that it is turned into a tree representation

Root	Answer Rel Ops
Intermediate branches	Intermediate Rel Ops
Leaves	Relations

Query Optimization

- When a query is parsed, we will assume that it is turned into a tree representation

$$(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$$


$\bowtie_{Staff.branchNo=Branch.branchNo}$
$\sigma_{position='Manager'}$
$\sigma_{city='London'}$
Staff
Branch

What is set of valid tree reorderings?

Query Optimization

What is the appropriate tree representation for:

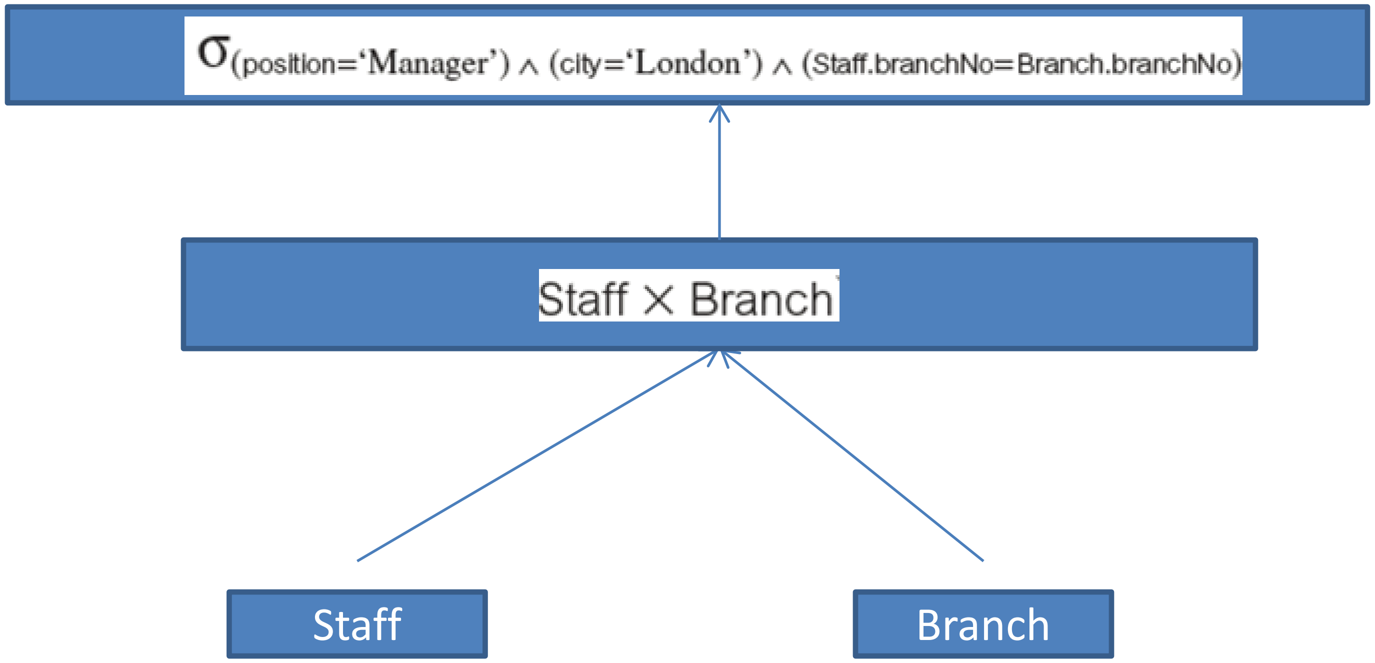
$$\sigma_{(\text{position}=\text{'Manager'}) \wedge (\text{city}=\text{'London'}) \wedge (\text{Staff.branchNo}=\text{Branch.branchNo})}(\text{Staff} \times \text{Branch})$$

Staff
Branch

Query Optimization

- What is appropriate representation for:

$$\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)}(Staff \times Branch)$$



$\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)}$
Staff × Branch
Staff
Branch