Understanding the Execution of Analytics Queries & Applications

MAS DSE 201

SQL as declarative programming

- SQL is a **declarative** programming language:
  - The developer's / analyst's query only describes **what** result she wants from the database
  - The developer does not describe the algorithm that the database will use in order to compute the result
- The database's optimizer automatically decides what is the most performant algorithm that computes the result of your SQL query
- "Declarative" and "automatic" have been the reason for the success and ubiquitous presence of database systems behind applications
  - Imagine trying to come up yourself with the algorithms that efficiently execute complex queries. (Not easy.)

2

What do you have to do to increase the performance of your db-backed app?

- Does declarative programming mean the developer does not have to think about performance?
  - After all, the database will automatically select the most performant algorithms for the developer's SQL queries
- **No, challenging cases force the A+ SQL developer / analyst to think and make choices, because…**
  - Developer decides which indices to build
  - Database may miss the best plan: Developer has to understand what plan was chosen and work around

3

Diagnostics

- You need to understand a few things about the performance of your query:

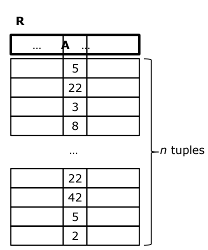
1. Will it benefit from indices? If yes, which are the useful indices?
2. Has the database chosen a hugely suboptimal plan?
3. How can I hack it towards the efficient way?

Boosting performance with indices (a short conceptual summary)

How/when does an index help? Running selection queries without an index

Consider a table R with *n* tuples and the selection query

SELECT *
FROM R
WHERE R.A = ?



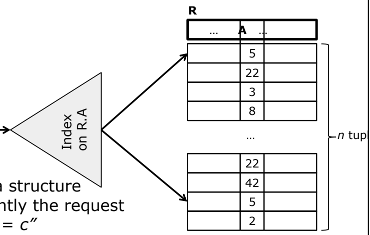
In the absence of an index the *Big-O* cost of evaluating an instance of this query is *O(n)* because the database will need to access the *n* tuples and check the condition R.A = <*provided value*>

How/when does an index help? Running selection queries with an index

Consider a table R with *n* tuples, an index on R.A
and assume that R.A has *m* distinct values.
We issue the same query and assume the database uses the index.

```
SELECT *
FROM R
WHERE R.A = ?
```

Example request: Return pointers to tuples with R.A = 5



An index on R.A is a data structure
that answers very efficiently the request
"find the tuples with R.A = *c*"
Then a query is answered in time *O(k)*
where k is the number of tuples with R.A = c.
Therefore the expected time to answer a selection query is *O(n/m)*

7

The mechanics of indices: How to create an index

How to create an index on R.A ?

After you have created table **R**, issue command
CREATE INDEX myIndexOnRA ON R(A)

How to remove the index you previously created ?

DROP INDEX myIndexOnRA

Exercise: Create and then drop an index on Students.first_name of the enrollment example

After you have created table **students**, issue command
CREATE INDEX students_first_name ON students(first_name)

DROP INDEX students_first_name

Primary keys get an index automatically

8

The mechanics of indices: How to use an index in a query

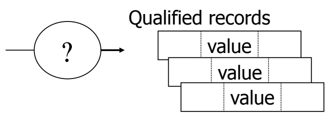
- You do **not** have to change your SQL queries in order to direct the database to use (or not use) the indices you created.
  - All you need to do is to create the index! That's easy…
- The database will decide automatically whether to use (or not use) a created index to answer your query.
- It is possible that you create an index *x* but the database may not use it if it judges that there is a better plan (algorithm) for answering your query, without using the index *x*.

9

Indexing will help any query step when the problem is…

Given condition on attribute find qualified records

Attr = value



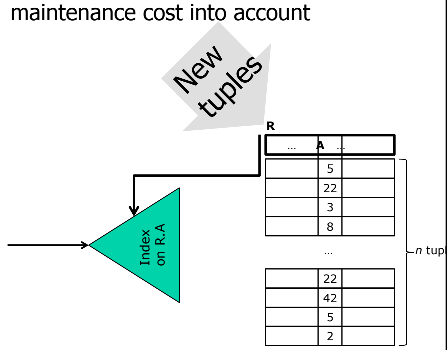
Condition may also be

- Attr>value
- Attr>=value

Indexing

- Data Stuctures used for quickly locating tuples that meet a specific type of condition
  - *Equality* condition: find Movie tuples where Director=*X*
  - Other conditions possible, eg, *range* conditions: find Employee tuples where Salary>40 AND Salary<50
- Many types of indexes. Evaluate them on
  - *Access* time
  - *Insertion* time
  - *Deletion* time
  - *Space* needed (esp. as it effects access time and or ability to fit in memory)

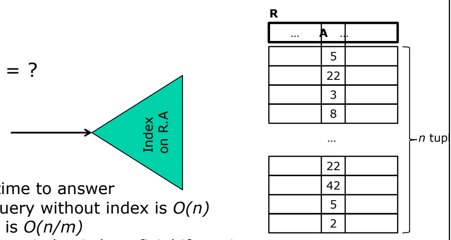
Should I build an index? In the presence of updates, the benefit of an index has to take maintenance cost into account



In OLAP it seems beneficial to create an index on R.A whenever m>1

Recall: Table R with *n* tuples, an index on R.A
and assume that R.A has *m* distinct values

SELECT *
FROM R
WHERE R.A = ?



The expected time to answer
the selection query without index is *O(n)*
and with index is *O(n/m)*
It appears that an index is beneficial if *m>1*
but if database stored in secondary storage you will need *m>>1*
because **the cost is blocks**!

To Index or Not to Index

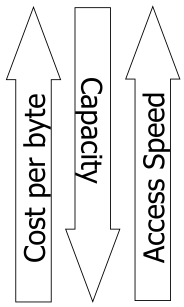
- Which queries can use indices and how?
- What will they do without an index?
  - Some surprisingly efficient algorithms that do not use indices

14

Understanding Storage and Memory

Memory Hierarchy

- Cache memory
  - On-chip and L2
  - Increasingly important
- RAM (controlled by db system)
  - Addressable space includes virtual memory but DB systems avoid it
- SSDs
  - Block-based storage
- Disk
  - Block
  - Preference to sequential access
- Tertiary storage for archiving
  - Tapes, jukeboxes, DVDs
  - Does not matter any more



16

Non-Volatile Storage is important to OLTP even when RAM is large

- Persistence important for transaction atomicity and durability
- Even if database fits in main memory changes have to be written in non-volatile storage
- Hard disk
- RAM disks w/ battery
- Flash memory

17

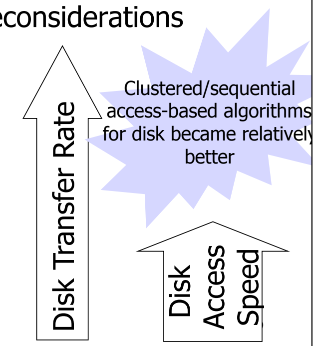
Peculiarities of storage mediums affect algorithm choice

- Block-based access:
  - Access performance: How many blocks were accessed
  - ~~How many objects~~
  - Flash is different on reading Vs writing
- Clustering for sequential access:
  - Accessing consecutive blocks costs less on disk-based systems
- We will only consider the effects of block access

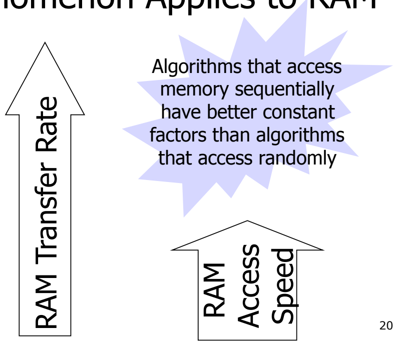
18

Moore's Law: Different Rates of Improvement Lead to Algorithm & System Reconsiderations

- Processor speed
- Main memory bit/$
- Disk bit/$
- RAM access speed
- Disk access speed
- Disk transfer rate

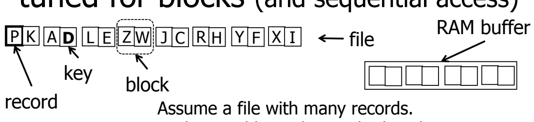


Moore's Law: Same Phenomenon Applies to RAM



20

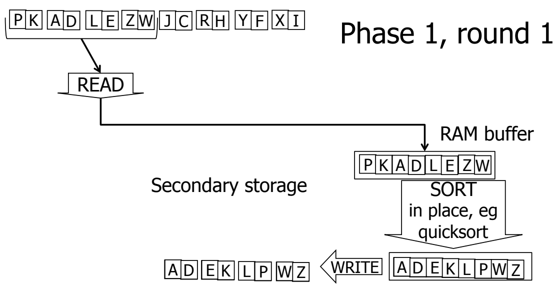
2-Phase Merge Sort: An algorithm tuned for blocks (and sequential access)



Assume a file with many records.
Each record has a key and other data.
For ppt brevity, the slide shows only the key of each record and not its data.
Assume each block has 2 records.
Assume RAM buffer fits 4 blocks (8 records)
In practice, expect many more records per block and many more records fitting in buffer.

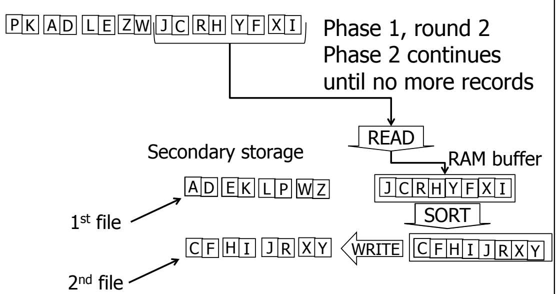
Problem: Sort the records according to the key.
Morale: What you learnt in algorithms and data structures is not always the best when we consider block-based storage

2-Phase Merge Sort



22

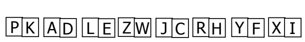
2-Phase Merge Sort



In practice, probably many more Phase 1 rounds and many respective output files

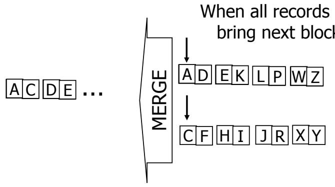
23

2-Phase Merge Sort



Phase 2

Assume #files < #blocks that fit in RAM buffer.
Fetch the first block of each file in RAM buffer.
Merge records and output.
When all records of a block have been output,
bring next block of same file



Improvement: Bring max number of blocks in memory.

24

2-Phase Merge Sort: Most files can be sorted in just 2 passes!

Assume

- *M* bytes of RAM buffer (eg, 8GB)
- *B* bytes per block (eg, 64KB for disk, 4KB for SSD)

Calculation:

- The assumption of Phase 2 holds when *#files* < *M/B*

=> there can be up to *M/B* Phase 1 rounds

- Each round can process up to *M* bytes of input data

=> 2-Phase Merge Sort can sort ***M²/B*** bytes

  – eg $(8GB)^2/64KB = (2^{33}B)^2 / 2^{16}B = 2^{50}B$ = 1PB

Horizontal placement of SQL data in blocks

Relations:

- Pack as many tuples per block
  – improves scan time
- Do not reclaim deleted records
- Utilize overflow records if relation must be sorted on primary key
- A novel generation of databases features column storage
  – to be discussed late in class

Sample relational database

Classes

id	name	number	date_code	start_time	end_time
1	Web stuff	CSE135	TuTh	2:00	3:20
2	Databases	CSE132A	TuTh	3:30	4:50
4	VLSI	CSE121	F	*null*	*null*

Enrollment

id	class	student	credits
1	1	1	4
2	1	2	3
3	4	3	4
4	1	3	3

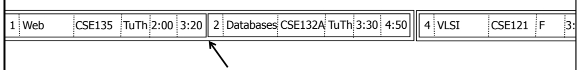
Students

id	pid	first_name	last_name
1	8888888	John	Smith
2	1111111	Mary	Doe
3	2222222	*null*	Chen

Pack maximum #records per block

Classes

id	name	number	date_code	start_time	end_time
1	Web	CSE135	TuTh	2:00	3:20
2	Databases	CSE132A	TuTh	3:30	4:50
4	VLSI	CSE121	F	*null*	*null*



"pack" each block with maximum # records

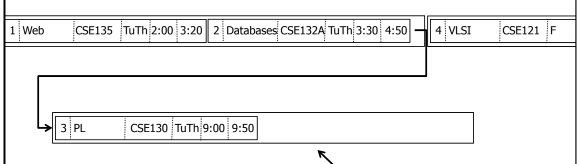
28

Utilize overflow blocks for insertions with "out of order" primary keys

Classes

id	name	number	date_code	start_time	end_time
1	Web	CSE135	TuTh	2:00	3:20
2	Databases	CSE132A	TuTh	3:30	4:50
3	**PL**	**CSE130**	**TuTh**	**9:00**	**9:50**
4	VLSI	CSE121	F	*null*	*null*

just inserted tuple



Overflow block

29

... back to Indices, with secondary storage in mind

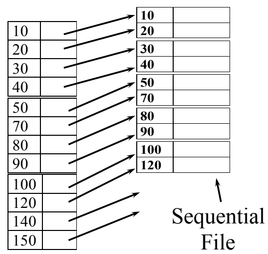
- Conventional indexes
  - As a thought experiment
- B-trees
  - The workhorse of most db systems
- Hashing schemes
  - Briefly covered
- Bitmaps
  - An analytics favorite

30

Terms and Distinctions

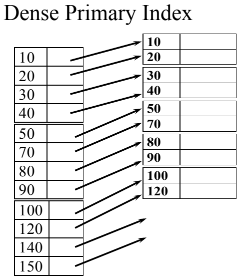
- *Primary* index
  - the index on the attribute (a.k.a. search key) that determines the sequencing of the table
- *Secondary* index
  - index on any other attribute
- *Dense* index
  - every value of the indexed attribute appears in the index
- *Sparse* index
  - many values do not appear

A Dense Primary Index

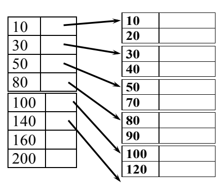


Dense and Sparse Primary Indexes

Dense Primary Index



Sparse Primary Index



Find the index record with largest value that is less or equal to the value we are looking.

+ can tell if a value exists without accessing file (consider projection)
+ better access to overflow records

+ less index space

more + and - in a while

<u>Sparse vs. Dense Tradeoff</u>

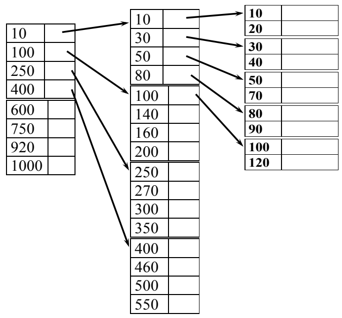
- <u>Sparse:</u> Less index space per record can keep more of index in memory
- <u>Dense:</u> Can tell if any record exists without accessing file

(Later:
- sparse better for insertions
- dense needed for secondary indexes)

33

Multi-Level Indexes

- Treat the index as a file and build an index on it
- "Two levels are usually sufficient. More than three levels are rare."
- Q: Can we build a dense second level index for a dense index ?

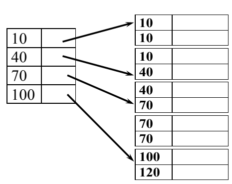


A Note on Pointers

- *Record pointers* consist of *block pointer* and position of record in the block
- Using the block pointer only, saves space at no extra accesses cost
- But a block pointer cannot serve as record identifier

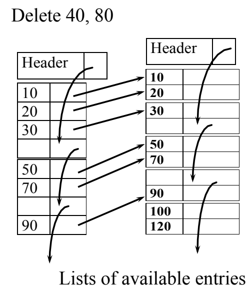
Representation of Duplicate Values in Primary Indexes

- Index may point to first instance of each value only



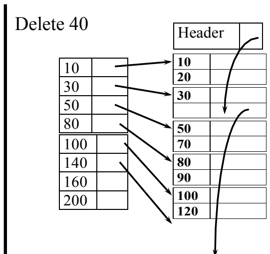
Deletion from Dense Index

- Deletion from dense primary index file with no duplicate values is handled in the same way with deletion from a sequential file
- Q: What about deletion from dense primary index with duplicates



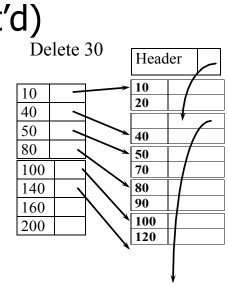
Deletion from Sparse Index

- if the deleted entry does not appear in the index do nothing



Deletion from Sparse Index (cont'd)

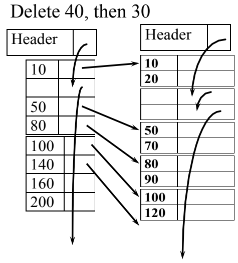
- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
  - comment: we could leave the deleted value in the index assuming that no part of the system may assume it still exists without checking the block



Deletion from Sparse Index (cont'd)

- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
- unless the next search key value has its own index entry. In this case delete the entry

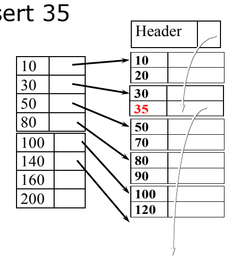
Delete 40, then 30



Insertion in Sparse Index

- if no new block is created then do nothing

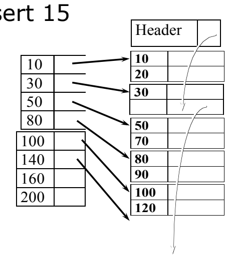
Insert 35



Insertion in Sparse Index

- if no new block is created then do nothing
- else create overflow record
  - Reorganize periodically
  - Could we claim space of next block?
  - How often do we reorganize and how much expensive it is?
  - B-trees offer convincing answers

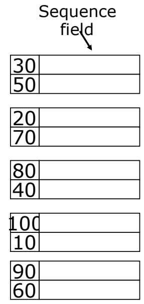
Insert 15



42

Secondary indexes

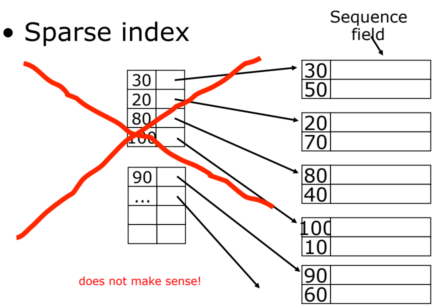
File not sorted on secondary search key



43

Secondary indexes

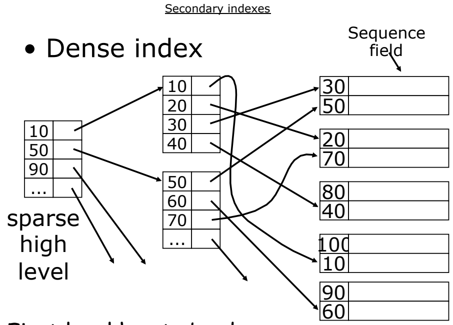
- Sparse index



44

Secondary indexes

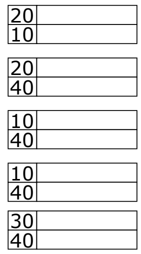
- Dense index



First level has to be dense, next levels are sparse (as usual)

45

Duplicate values & secondary indexes

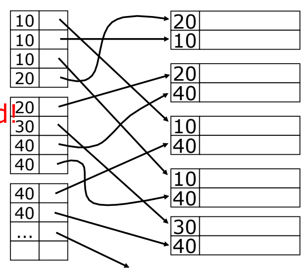


46

Duplicate values & secondary indexes

one option...

Problem:
excess overhead!
- disk space
- search time

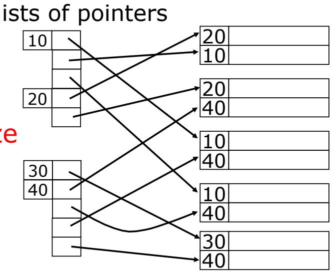


47

Duplicate values & secondary indexes

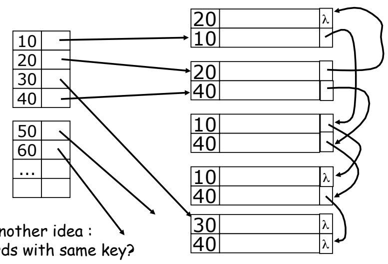
nother option: lists of pointers

Problem:
variable size
records in
index!



48

Duplicate values & secondary indexes

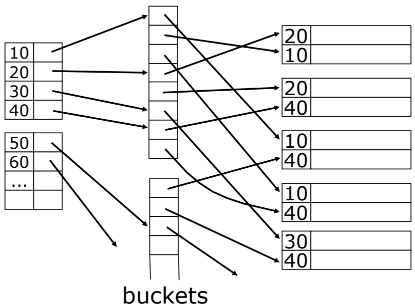


Yet another idea :
Chain records with same key?

Problems:
• Need to add fields to records, messes up maintenance
• Need to follow chain to know records

49

Duplicate values & secondary indexes



50

Why "bucket" + record pointers is useful

• Enables the processing of queries working with pointers only.
• Very common technique in Information Retrieval

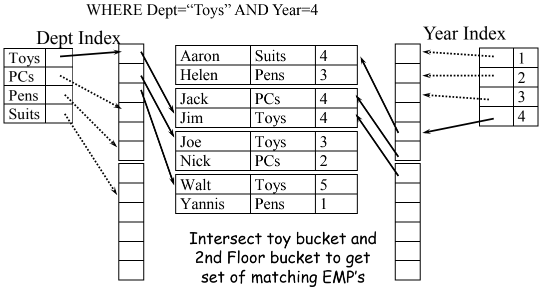
Indexes	Records
Name: primary	EMP (name,dept,year,...)
Dept: secondary	
Year: secondary	

51

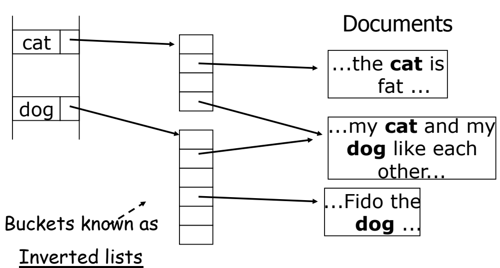
Advantage of Buckets: Process Queries Using Pointers Only

Find employees of the Toys dept with 4 years in the company

SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4



This idea used in text information retrieval



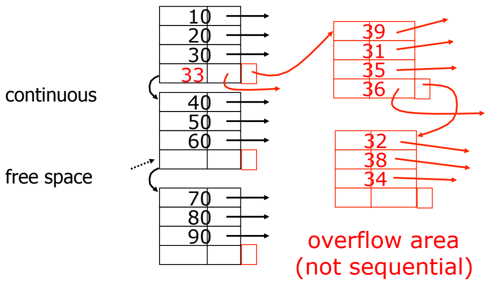
53

Summary of Indexing So Far

- Basic topics in conventional indexes
  - multiple levels
  - sparse/dense
  - duplicate keys and buckets
  - deletion/insertion similar to sequential files
- Advantages
  - simple algorithms
  - index is sequential file
- Disadvantages
  - eventually sequentiality is lost because of overflows, reorganizations are needed

Example

Index (sequential)



55

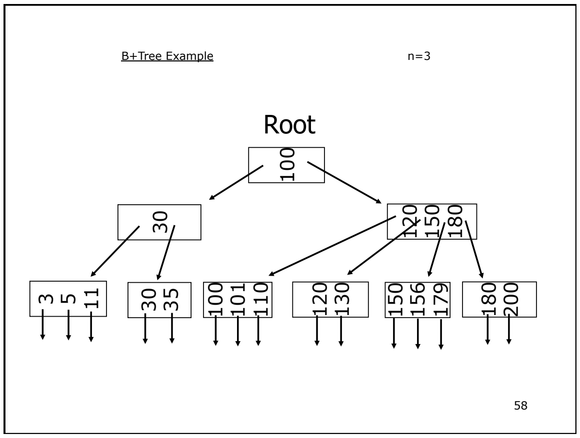
Outline:

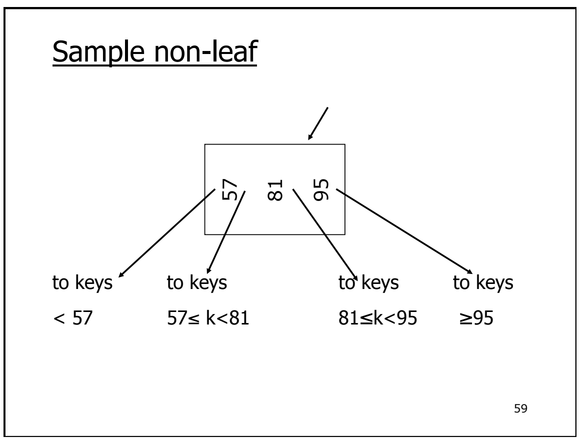
- Conventional indexes
- B-Trees $\Rightarrow$ NEXT
- Hashing schemes

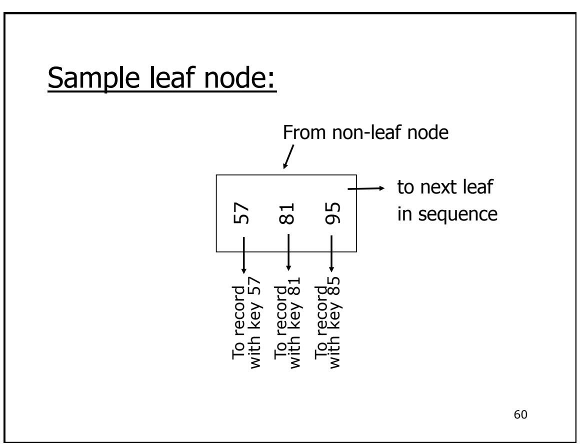
56

- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

57

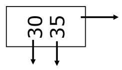

B+Tree Example
n=3
Root
100
30
120 150 180
3 5 11
30 35
100 101 110
120 130
150 156 179
180 200
58

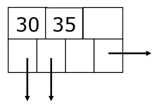

Sample non-leaf
57 81 95
to keys
< 57
to keys
57≤ k<81
to keys
81≤k<95
to keys
≥95
59


Sample leaf node:
From non-leaf node
57 81 95
to next leaf
in sequence
To record with key 57
To record with key 81
To record with key 85
60

In textbook's notation n=3

Leaf:





Non-leaf:

61

Size of nodes:
- n+1 pointers
- n keys (fixed)

62

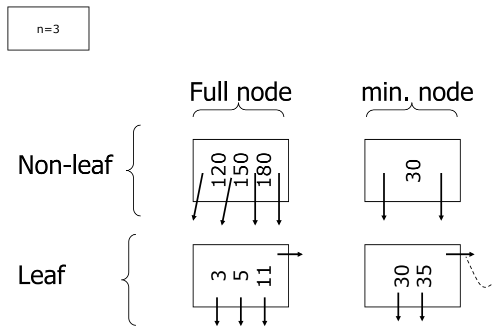
Non-root nodes have to be at least half-full

- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

63



64

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"

65

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

	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+1	n	$\lfloor(n+1)/2\rfloor$	$\lfloor(n+1)/2\rfloor$
Root	n+1	n	1	1

66

Insert into B+tree

(a) simple case
- space available in leaf

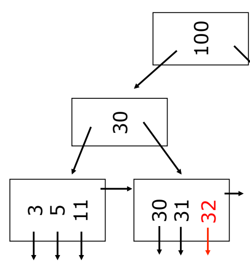
(b) leaf overflow

(c) non-leaf overflow

(d) new root

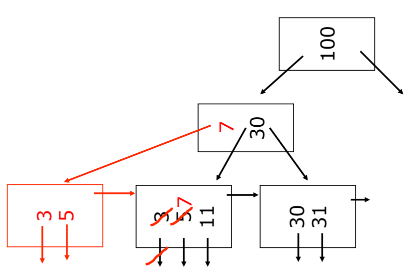
(a) Insert key = 32

n=3



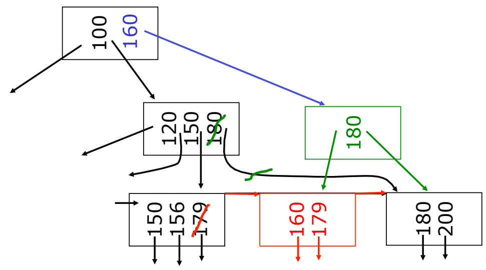
(a) Insert key = 7

n=3



(c) Insert key = 160

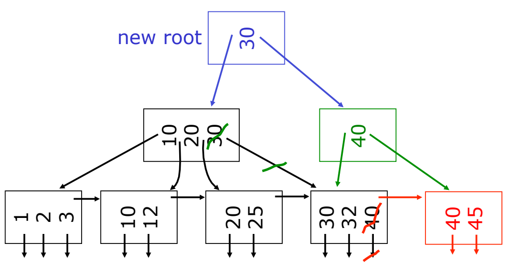
n=3



70

(d) New root, insert 45

n=3



71

Deletion from B+tree

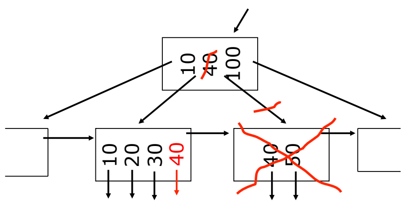
(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf

72

(b) Coalesce with sibling

– Delete 50

n=4

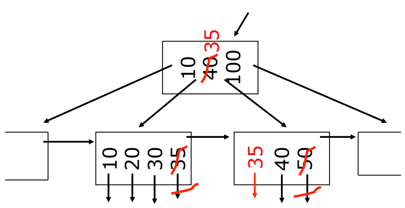


73

(c) Redistribute keys

– Delete 50

n=4

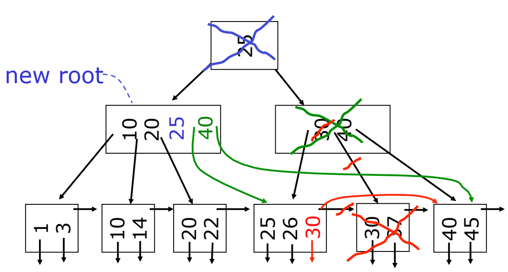


74

(d) Non-leaf coalese

– Delete 37

n=4



75

B+tree deletions in practice

– Often, coalescing is not implemented
  – Too hard and not worth it!

76

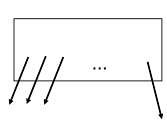
Is LRU a good policy for B+tree buffers?

→ Of course not!
→ Should try to keep root in memory
at all times
(and perhaps some nodes from second level)

77

Hardware+ indexing problem:

For B+tree, how large should *n* be?



n is number of keys / node

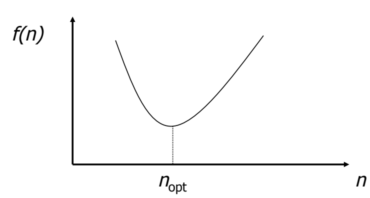
78

Assumptions

- You have the right to set the block size for the disk where a B-tree will reside.
- Compute the optimum page size *n* assuming that
  - The items are 4 bytes long and the pointers are also 4 bytes long.
  - Time to read a node from disk is *12+.003n*
  - Time to process a block in memory is unimportant
  - B+tree is full (I.e., every page has the maximum number of items and pointers

➛Can get:

f(n) = time to find a record



80

➛ FIND n_{opt} by $f'(n) = 0$

Answer should be n_{opt} = "few hundred"

➛ What happens to n_{opt} as

- Disk gets faster?
- CPU get faster?

81

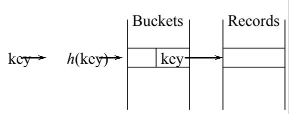
Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B+ trees
- Hashing schemes --> Next
- Bitmap indices

82

Hashing

- hash function *h*(key) returns address of bucket
- if the keys for a specific hash value do not fit into one page the bucket is a linked list of pages



Example hash function

- Key = 'x_1 x_2 … x_n' *n* byte character string
- Have *b* buckets
- h: add $x_1 + x_2 + \ldots\ldots x_n$
  - compute sum modulo *b*

84

- This may not be best function …
- Read Knuth Vol. 3 if you really need to select a good function.

Good hash function: ☞ Expected number of keys/bucket is the same for all buckets

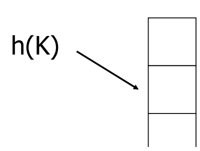
85

Within a bucket:

- Do we keep keys sorted?
  - Yes, if CPU time critical & Inserts/Deletes not too frequent

86

Next: example to illustrate inserts, overflows, deletes

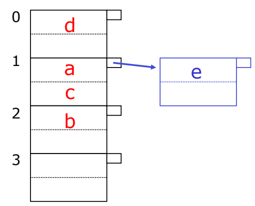


87

EXAMPLE 2 records/bucket

INSERT:
h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0

h(e) =
1

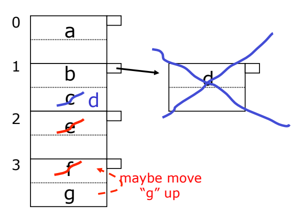


88

EXAMPLE: deletion

Delete:
e
f

c



89

Rule of thumb:

- Try to keep space utilization between 50% and 80%

Utilization = # keys used / total # keys that fit

- If < 50%, wasting space
- If > 80%, overflows significant depends on how good hash function is & on # keys/bucket

90

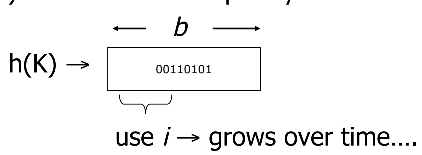
How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

91

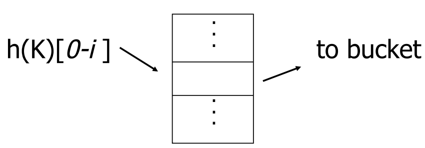
Extensible hashing: two ideas

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

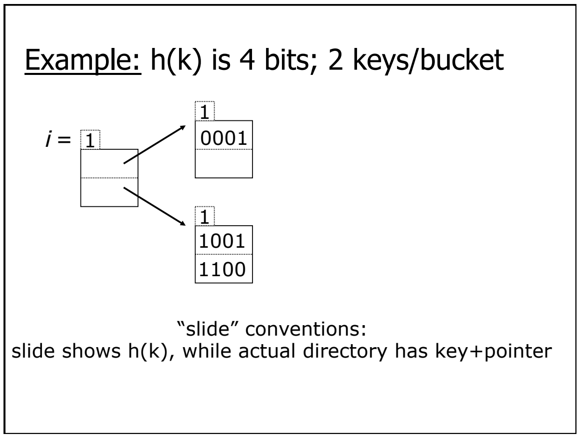


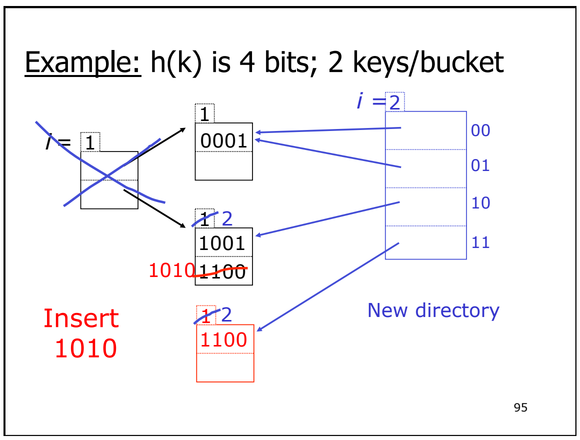
92

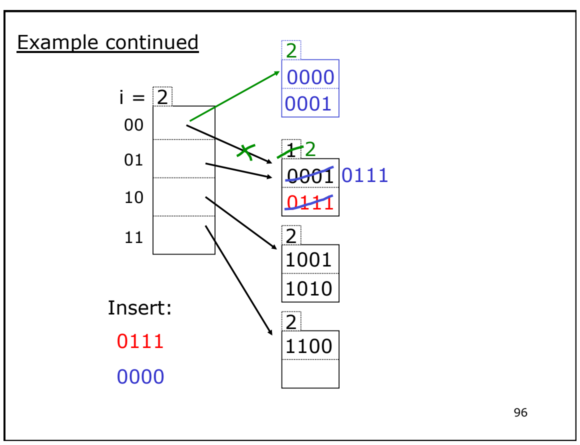
(b) Use directory



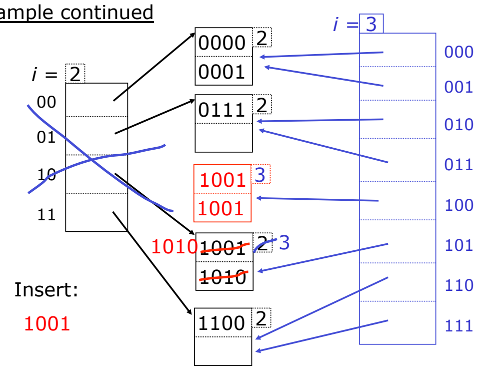
93


Example: h(k) is 4 bits; 2 keys/bucket
i = 1
1
0001
1
1001
1100
"slide" conventions:
slide shows h(k), while actual directory has key+pointer


Example: h(k) is 4 bits; 2 keys/bucket
i = 1
1
0001
1 2
1001
1010 1100
Insert
1010
1 2
1100
i = 2
00
01
10
11
New directory
95


Example continued
i = 2
00
01
10
11
2
0000
0001
1 2
0001 0111
0111
2
1001
1010
2
1100
Insert:
0111
0000
96

Example continued



97

Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

98

Deletion example:

- Run thru insert example in reverse!

99

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

(Now it fits, now it does not)

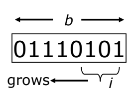
100

Linear hashing

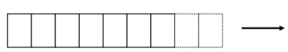
- Another dynamic hashing scheme

Two ideas:

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



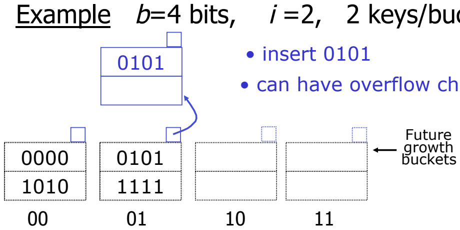
(b) File grows linearly



101

Example *b*=4 bits, *i* =2, 2 keys/bucket

- insert 0101
- can have overflow chains!



m = 01 (max used block)

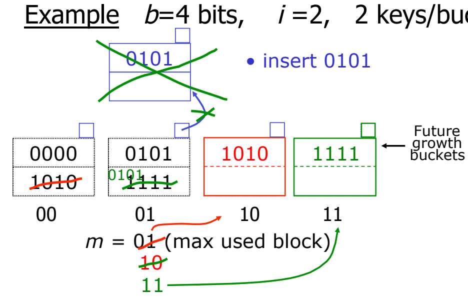
Rule

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

102

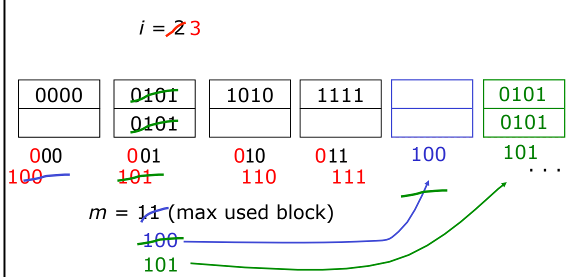
Example b=4 bits, i =2, 2 keys/bucket

• insert 0101



103

Example Continued: How to grow beyond this?



104

☛ When do we expand file?

- Keep track of: $\frac{\#\text{used slots (incl. overflow)}}{\#\text{total slots in primary buckets}} = U$

  equiv, $\frac{\#\text{(indexed key ptr pairs)}}{\#\text{total slots in primary buckets}}$

- If U > threshold then increase m (and i, when m reaches 2^i)

105

Summary

Linear Hashing

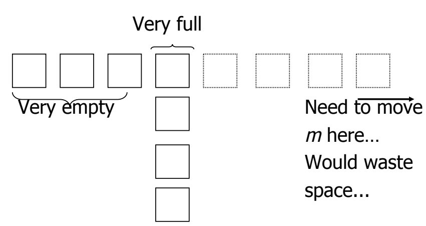
⊕ Can handle growing files
  - with less wasted space
  - with no full reorganizations

⊕ No indirection like extensible hashing

⊖ Can still have overflow chains

106

Example: BAD CASE



107

Summary

Hashing
- How it works
- Dynamic hashing
  - Extensible
  - Linear

108

Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access

109

Indexing vs Hashing

- Hashing good for probes given key

  e.g., SELECT …
  FROM R
  WHERE R.A = 5

110

Indexing vs Hashing

- INDEXING (Including B Trees) good for
  Range Searches:
  e.g., SELECT
  FROM R
  WHERE R.A > 5

111

Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)

  → defines candidate key

  - Drop INDEX name

112

Note CANNOT SPECIFY TYPE OF INDEX
(e.g. B-tree, Hashing, …)
OR PARAMETERS
(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

113

Note ATTRIBUTE LIST ⇒ MULTIKEY INDEX (next)

e.g., CREATE INDEX foo ON R(A,B,C)

114

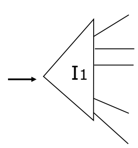
Multi-key Index

Motivation: Find records where
DEPT = "Toy" AND SAL > 50k

115

Strategy I:

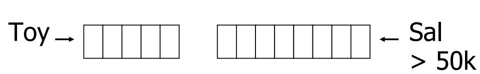
- Use one index, say Dept.
- Get all Dept = "Toy" records and check their salary



116

Strategy II:

- Use 2 Indexes; Manipulate Pointers

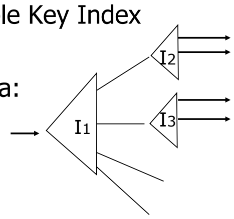


117

Strategy III:

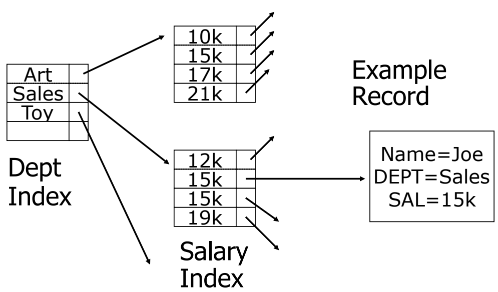
- Multiple Key Index

One idea:



118

Example



119

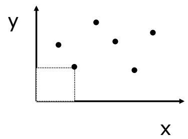
For which queries is this index good?

- ☐ Find RECs Dept = “Sales” ∧ SAL=20k
- ☐ Find RECs Dept = “Sales” ∧ SAL ≥ 20k
- ☐ Find RECs Dept = “Sales”
- ☐ Find RECs SAL = 20k

120

Interesting application:

- Geographic Data



DATA:
<X_1,Y_1, Attributes>
<X_2,Y_2, Attributes>
⋮

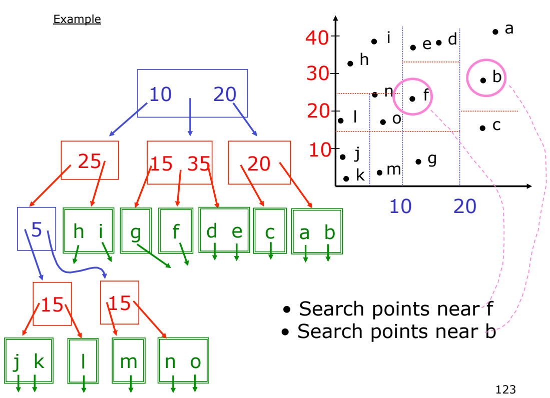
121

Queries:

- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

122

Example



- Search points near f
- Search points near b

123

Queries

- Find points with Yi > 20
- Find points with Xi < 5
- Find points "close" to i = <12,38>
- Find points "close" to b = <7,24>

124

- Many types of geographic index structures have been suggested
  - Quad Trees
  - R Trees

125

Outline/summary

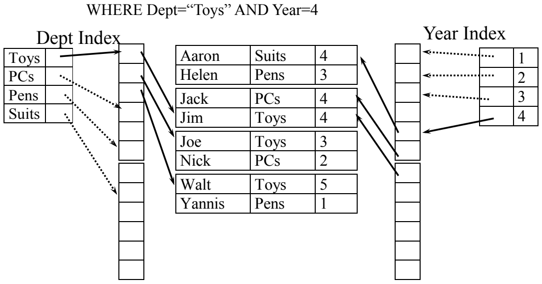
- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B+ trees
- Hashing schemes
- Bitmap indices --> Next

126

Revisit: Processing queries without accessing records until last step

Find employees of the Toys dept with 4 years in the company

SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4



Bitmap indices: Alternate structure, heavily used in OLAP

Assume the tuples of the Employees table are ordered.

Conceptually only!

Dept Index

Toys	00011010
PCs	00100100
Pens	01000001
Suits	10000000

Aaron	Suits	4
Helen	Pens	3
Jack	PCs	4
Jim	Toys	4
Joe	Toys	3
Nick	PCs	2
Walt	Toys	1
Yannis	Pens	1

Year Index

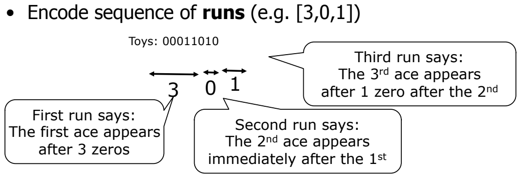
00000011	1
00000100	2
01001000	3
10110000	4

+ Find even more quickly intersections and unions (e.g., Dept="Toys" AND Year=4)
? Seems it needs too much space -> We'll do compression
? How do we deal with insertions and deletions -> Easier than you think

128

Compression, with Run-Length Encoding

- Naive solution needs *mn* bits, where *m* is #distinct values and *n* is #tuples
- But there is just *n* 1's=> let's utilize this
- Encode sequence of **runs** (e.g. [3,0,1])



129

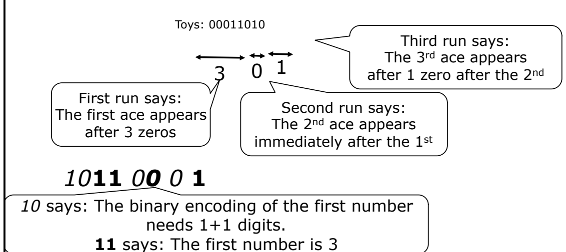
Byte-Aligned Run Length Encoding

Next key intuition: Spend fewer bits for smaller numbers

Consider the run
5, 200, 17
In binary it is
101, 11000100, 10001

A binary number of up to 7 bits => 1 byte
A binary number of up to 14 bits => 2 bytes
…
Use the first bit of each byte to denote if it is the last one of a number
0**0000101**, 1**0000001**, 0**1000100**, 0**0010001** 130

Bit-aligned 2nlog*m* Compression (simple version)



2nlog m compression

- Example
- Pens: 01000001
- Sequence [1,5]
- Encoding: *0***1***110***101**

132

Insertions and deletions & miscellaneous engineering

- Assume tuples are inserted in order
- Deletions: Do nothing
- Insertions: If tuple *t* with value *v* is inserted, add one more run in *v*'s sequence (compact bitmap)

133

Summing Up…

We discussed how the database stores data + basic algorithms

- Sorting
- Indexing

How are they used in query processing?

134

Query Processing Notes

What happens when a query is processed and how to find out

Query Processing

- The query processor turns user queries and data modification commands into a query plan - a sequence of operations (or algorithm) on the database
  - from high level queries to low level commands
- Decisions taken by the query processor
  - Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
  - For each algebraic operator what algorithm should we use to run the operator?
  - How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

The differences between good plans and plans can be huge Example

Select B,D
From R,S
Where R.A = “c” ∧ S.E = 2 ∧ 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 query eventually?

One idea →

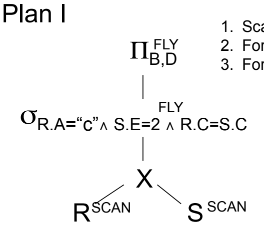
- Scan relations
- Do Cartesian product (literally produce all combinations of FROM clause tuples)
- Select tuples (WHERE)
- Do projection (SELECT)

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
	.					
	.					
Bingo! Got one... →	C	2	10	10	x	2
	.					
	.					

Relational Plan:

Ex: Plan I

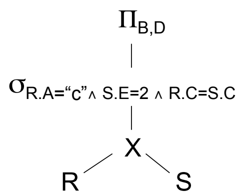


1. Scan R
2. For each tuple r of R scan S
3. For each (r,s), where s in S select and project on the fly

OR: $\Pi^{FLY}_{B,D}\ [\ \sigma^{FLY}_{R.A=\text{“c”} \wedge S.E=2 \wedge R.C = S.C}\ (R^{SCAN} X\ S^{SCAN})]$

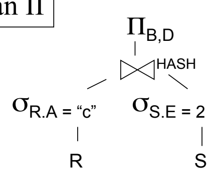
"FLY" and "SCAN" are the defaults

Ex: Plan I



Another idea:

Plan II



Scan R and S, perform on the fly selections,
do join using a hash structure, project

R

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

σ (R)

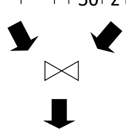
A	B	C
c	2	10

σ(S)

C	D	E
10	x	2
20	y	2
30	z	2

S

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



Plan III

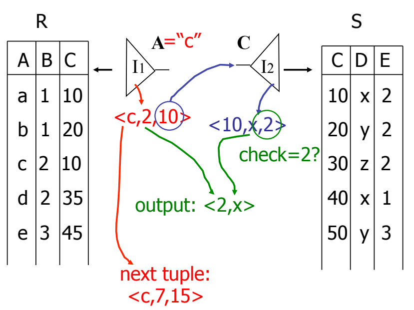
Use R.A and S.C Indexes

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

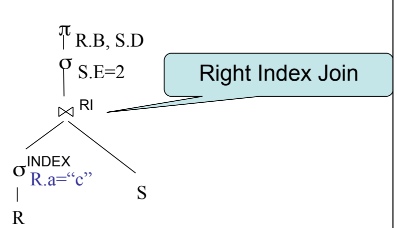
(2) For each R.C value found, use S.C index to find matching join tuples

(3) Eliminate join tuples S.E ≠ 2

(4) Project B,D attributes



Algebraic Form of Plan

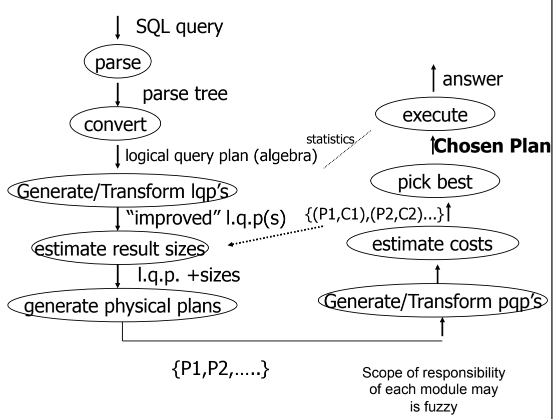


From Query To Optimal Plan

- Complex process
- Algebra-based logical and physical plans
- Transformations
- Evaluation of multiple alternatives

Issues in Query Processing and Optimization

- Generate Plans
  - employ efficient execution primitives for computing relational algebra operations
  - systematically transform expressions to achieve more efficient combinations of operators
- Estimate Cost of Generated Plans
  - Statistics, which are reported



Algebraic Operators: A Bag version

- *Union of R and S*: a tuple t is in the result as many times as the sum of the number of times it is in R plus the times it is in S
- *Intersection of R and S*: a tuple t is in the result the minimum of the number of times it is in R and S
- *Difference of R and S*: a tuple t is in the result the number of times it is in R minus the number of times it is in S
- $\delta(R)$ converts the bag *R* into a set
  - SQL's *R UNION S* is really $\delta(R \cup S)$
- **Example:** Let *R={A,B,B}* and *S={C,A,B,C}*.Describe the union, intersection and difference...

Extended Projection

- project π_A , A is attribute list
  - The attribute list may include $x \rightarrow y$ in the list *A* to indicate that the attribute *x* is renamed to *y*
  - Arithmetic, string operators and scalar functions on attributes are allowed. For example,
    - $a+b \rightarrow x$ means that the sum of *a* and *b* is renamed into *x*.
    - $c||d \rightarrow y$ concatenates the result of *c* and *d* into a new attribute named *y*
- The result is computed by considering each tuple in turn and constructing a new tuple by picking the attributes names in *A* and applying renamings and arithmetic and string operators
- **Example:**

Products and Joins

- *Product of R and S (R×S)*:
  - If an attribute named *a* is found in both schemas then rename one column into *R.a* and the other into *S.a*
  - If a tuple *r* is found *n* times in *R* and a tuple *s* is found *m* times in *S* then the product contains *nm* instances of the tuple *rs*
- Joins
  - *Natural Join* $R \bowtie S = \pi_A \sigma_C(R \times S)$ where
    - *C* is a condition that equates all common attributes
    - *A* is the concatenated list of attributes of *R* and *S* with no duplicates
    - you may view tha above as a rewriting rule
  - *Theta Join*
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

- $\gamma_{GroupByList;\ aggrFn1 \rightarrow attr1,\ \ldots, aggrFnN \rightarrow attrN}$
- Conceptually, grouping leads to nested tables and is immediately followed by functions that aggregate the nested table
- Example: $\gamma_{Dept;\ AVG(Salary) \rightarrow AvgSal,\ \ldots,\ SUM(Salary) \rightarrow SalaryExp}$

Find the average salary for each department

```
SELECT Dept, AVG(Salary) AS AvgSal,
 SUM(Salary) AS SalaryExp
FROM Employee
GROUP-BY Dept
```

Employee

Name	Dept	Salary
Joe	Toys	45
Nick	PCs	50
Jim	Toys	35
Jack	PCs	40

Dept	Nested Table Name Salary
Toys	Joe 45 Jim 35
PCs	Nick 50 Jack 40

Dept	AvgSal	SalaryExp
Toys	40	80
PCs	45	90

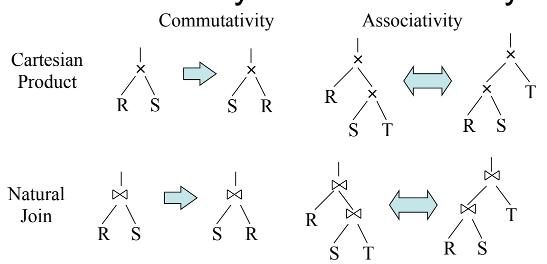
Sorting and Lists

- SQL and algebra results are ordered
- Could be non-deterministic or dictated by SQL ORDER BY, algebra τ
- $\tau_{OrderByList}$
- A result of an algebraic expression o(exp) is ordered if
  - If o is a τ
  - If o retains ordering of exp and exp is ordered
    - Unfortunately this depends on implementation of o
  - If o creates ordering
  - Consider that leaf of tree may be SCAN(R)

Relational algebra optimization

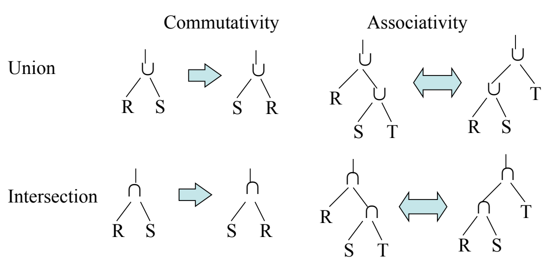
- Transformation rules (preserve equivalence)
- A quick tour

Algebraic Rewritings: Commutativity and Associativity



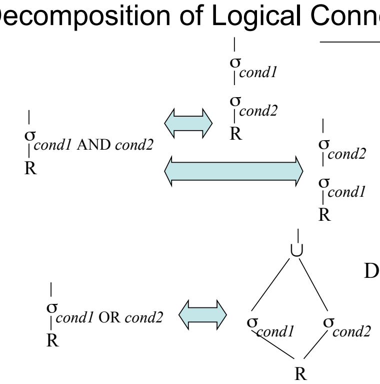
Question 1: Do the above hold for both sets and bags?
Question 2: Do commutativity and associativity hold for arbitrary Theta Joins?

Algebraic Rewritings: Commutativity and Associativity (2)



Question 1: Do the above hold for both sets and bags?
Question 2: Is difference commutative and associative?

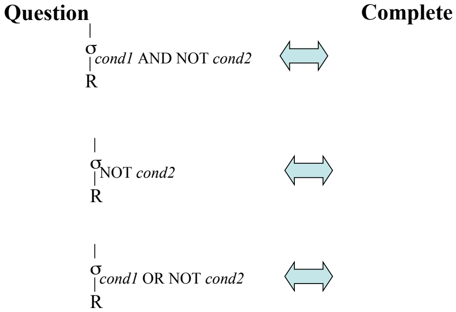
Algebraic Rewritings for Selection: Decomposition of Logical Connectives



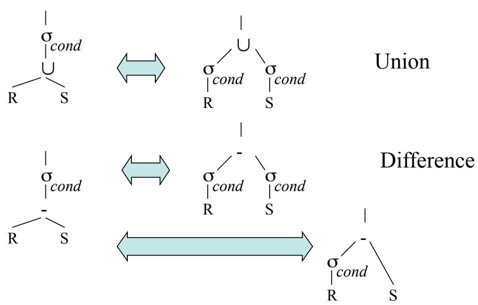
Does it apply to bags?

Algebraic Rewritings for Selection: Decomposition of Negation

Question **Complete**

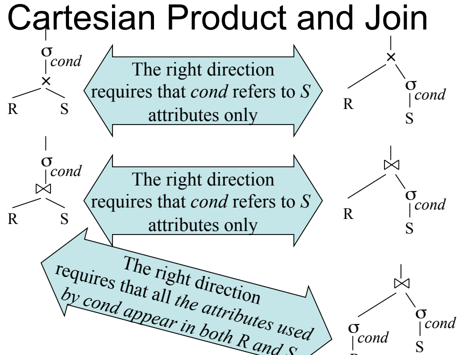


Pushing the Selection Thru Binary Operators: Union and Difference



Exercise: Do the rule for intersection

Pushing Selection thru Cartesian Product and Join



Exercise: Do the rule for theta join

Rules: π,σ combined

Let x = subset of R attributes

z = attributes in predicate P

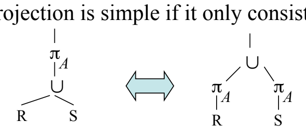
(subset of R attributes)

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

(in red: π_x struck out and replaced by π_{xz})

Pushing Simple Projections Thru Binary Operators

A projection is simple if it only consists of an attribute list



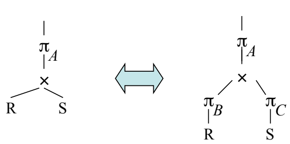
Union

Question 1: Does the above hold for both bags and sets?

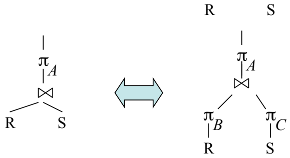
Question 2: Can projection be pushed below intersection and difference?

Answer for both bags and sets.

Pushing Simple Projections Thru Binary Operators: Join and Cartesian Product



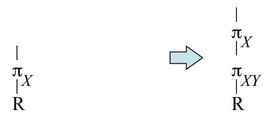
Where *B* is the list of *R* attributes that appear in *A*. Similar for *C*.



Question: What is *B* and *C* ?

Exercise: Write the rewriting rule that pushes projection below theta join.

Projection Decomposition



Derived Rules: σ + $\bowtie$ combined

More Rules can be Derived:

$\sigma_{p \wedge q}$ (R $\bowtie$ S) =

$\sigma_{p \wedge q \wedge m}$ (R $\bowtie$ S) =

$\sigma_{p \vee q}$ (R $\bowtie$ S) =

p only at **R**, **q** only at **S**, **m** at both **R** and **S**

--> Derivation for first one:

$\sigma_{p \wedge q}$ (R $\bowtie$ S) =

σ_p [σ_q (R $\bowtie$ S)] =

σ_p [R $\bowtie$ σ_q (S)] =

[σ_p (R)] $\bowtie$ [σ_q (S)]

Which are always "good" transformations?

- □ $\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$
- □ $\sigma_p(R \bowtie S) \rightarrow [\sigma_p(R)] \bowtie S$
- □ $R \bowtie S \rightarrow S \bowtie R$
- □ $\pi_x[\sigma_p(R)] \rightarrow \pi_x\{\sigma_p[\pi_{xz}(R)]\}$

<u>In textbook:</u> more transformations

- Eliminate common sub-expressions
- Other operations: duplicate elimination

<u>Bottom line:</u>

- No transformation is <u>always</u> good at the l.q.p level
- Usually good
  - early selections
  - elimination of cartesian products
  - elimination of redundant subexpressions
- Many transformations lead to "promising" plans
  - Commuting/rearranging joins
  - In practice too "combinatorially explosive" to be handled as rewriting of l.q.p.

Algorithms for Relational Algebra Operators

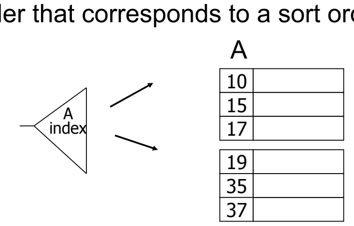
- Three primary techniques
  - Sorting
  - Hashing
  - Indexing
- Three degrees of difficulty
  - data small enough to fit in memory
  - too large to fit in main memory but small enough to be handled by a “two-pass” algorithm
  - so large that “two-pass” methods have to be generalized to “multi-pass” methods (quite unlikely nowadays)

The dominant cost of operators running on disk:

- Count # of disk blocks that must be read (or written) to execute query plan

Clustering index

Index that allows tuples to be read in an order that corresponds to a sort order



Clustering can radically change cost

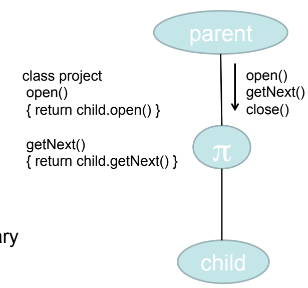
- Clustered relation

  R1 R2 R3 R4   R5 R5 R7 R8 …..

- Clustering index

Pipelining can radically change cost

- Interleaving of operations across multiple operators
- Smaller memory footprint, fewer object allocations
- Operators support:
  - open()
  - getNext()
  - close()
- Simple for unary
- Pipelined operation for binary discussed along with physical operators



Example R1 ⋈ R2 over common attribute C

First we will see main memory-based implementations

- Iteration join (conceptually – without taking into account disk block issues)
- For each tuple of left argument, re-scan the right argument

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

Also called "nested loop join" in some databases (eg Postgres)

- Join with index (Conceptually)
  - alike iteration join but right relation accessed with index

Assume R2.C index

```
For each r ∈ R1 do
  [ X ← index (R2, C, r.C)
     for each s ∈ X do
          output r,s pair]
```

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

- 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
```

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

i	R1{i}.C	R2{j}.C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

- Hash join, hashing both sides (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

Variation: Hash one side only

Algorithm

(1) Hash R1 tuples into G buckets

(2) For each tuple r2 or R2

find i=hash(r2)

match r2 with tuples in Gi

What's the benefit in hashing both sides?
Wait till we discuss hash joins on secondary storage…

Disk-oriented Cost Model

- There are *M* main memory buffers.
  - Each buffer has the size of a disk block
- The input relation is read one block at a time.
- The cost is the number of blocks read.
- (Applicable to Hard Disks:) If *B* consecutive blocks are read the cost is *B/d*.
- The output buffers are not part of the *M* buffers mentioned above.
  - *Pipelining* allows the output buffers of an operator to be the input of the next one.
  - We do not count the cost of writing the output.

Notation

- $B(R)$ = number of blocks that R occupies
- $T(R)$ = number of tuples of R
- $V(R,[a_1, a_2, \ldots, a_n])$ = number of distinct tuples in the projection of R on $a_1, a_2, \ldots, a_n$

One-Pass Main Memory Algorithms for Unary Operators

- Assumption: Enough memory to keep the relation
- Projection and selection:
  - Scan the input relation *R* and apply operator one tuple at a time
  - Incremental cost of "on the fly" operators is 0
- Duplicate elimination and aggregation
  - create one entry for each group and compute the aggregated value of the group
  - it becomes hard to assume that CPU cost is negligible
    - main memory data structures are needed

One-Pass Nested Loop Join

- Assume $B(R)$ is less than M
- Tuples of *R* should be stored in an efficient lookup structure
- **Exercise:** Find the cost of the algorithm below

```
for each block Br of R do
  store tuples of Br in main memory
for each each block Bs of S do
  for each tuple s of Bs
    join tuples of s with matching tuples of R
```

A variation where the inner side is organized into a hash (hash join in some databases)

```
for each block Br of R do
  store tuples of Br in main memory
             hash buckets G1,…, Gn
for each each block Bs of S do
  for each tuple s of Bs
     find h=hash(s)
     join s with matching tuples in Gh
```

Generalization of Nested-Loops

```
for each chunk of M-1 blocks Br of R do
  store tuples of Br in main memory
  for each each block Bs of S do
    for each tuple s of Bs
      join tuples of s with matching tuples of R
```

Exercise: Compute cost

Simple Sort-Merge Join

- Assume natural join on *C*
- Sort *R* on *C* using the two-phase multiway merge sort
  - if not already sorted
- Sort *S* on *C*
- Merge (opposite side)
  - assume two pointers `Pr,Ps` to tuples on disk, initially pointing at the start
  - sets `R’, S’` in memory
- Remarks:
  - Very low average memory requirement during merging (but no guarantee on how much is needed)
  - **Cost:**

```
while Pr!=EOF and Ps!=EOF
  if *Pr[C] == *Ps[C]
    do_cart_prod(Pr,Ps)
  else if *Pr[C] > *Ps[C]
    Ps++
  else if *Ps[C] > *Pr[C]
    Pr++

function do_cart_prod(Pr,Ps)
  val=*Pr[C]
  while *Pr[C]==val
    store tuple *Pr in set R’
  while *Ps[C]==val
    store tuple *Ps in set S’;
  output cartesian product
      of R’ and S’
```

Efficient Sort-Merge Join

- Idea: Save two disk I/O's per block by combining the second pass of sorting with the ``merge".
- Step 1: Create sorted sublists of size *M* for *R* and *S*
- Step 2: Bring the first block of each sublist to a buffer
  - assume no more than *M* sublists in all
- Step 3:Repeatedly find the least *C* value *c* among the first tuples of each sublist. Identify all tuples with join value *c* and join them.
  - When a buffer has no more tuple that has not already been considered load another block into this buffer.

Efficient Sort-Merge Join Example

R

C	*RA*
1	r_1
2	r_2
3	r_3
...	...
20	r_{20}

S

C	*SA*
1	s_1
...	...
5	s_5
16	s_{16}
...	...
20	s_{20}

Assume that after first phase of multiway sort we get 4 sublists, 2 for *R* and 2 for *S*.
Also assume that each block contains two tuples.

R

3 7	8 10	11 13	14 16	17 18
1 2	4 5	6 9	12 15	19 20

S

1 3	5 17	
2 4	16 18	19 20

Sort and Merge Join are typically separate operators

- Modularity
  - The sorting needed by join is no different than the sorting needed by ORDER BY
- May be only one side or no side needs sorting

Two-Pass Hash-Based Algorithms

- General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value.
  - If there are *M* buffers pick *M-1* as the number of hash buckets
- Example: Duplicate Elimination
  - Phase 1: Hash each tuple of each input block into one of the *M-1* bucket/buffers. When a buffer fills save to disk.
  - Phase 2: For each bucket:
    - load the bucket in main memory,
    - treat the bucket as a small relation and eliminate duplicates
    - save the bucket back to disk.
  - **Catch:** Each bucket has to be less than *M.*
  - **Cost:**

Hash-Join Algorithms

- Assuming natural join, use a hash function that
  - is the same for both input arguments *R* and *S*
  - uses only the join attributes
- Phase 1: Hash each tuple of *R* into one of the *M-1* buckets R_i and similar each tuple of *S* into one of S_i
- Phase 2: For *i=1…M-1*
  - load R_i and S_i in memory
  - join them and save result to disk
- **Question:** What is the maximum size of buckets?
- **Question:** Does hashing maintain sorting?

Index-Based Join: The Simplest Version

Assume that we do natural join of *R(A,B)* and *S(B,C)* and there's an index on *S*

```
for each Br in R do
  for each tuple r of Br with B value b
     use index of S to find
               tuples {s1 ,s2 ,...,sn} of S with
B=b
     output {rs1 ,rs2 ,...,rsn}
```

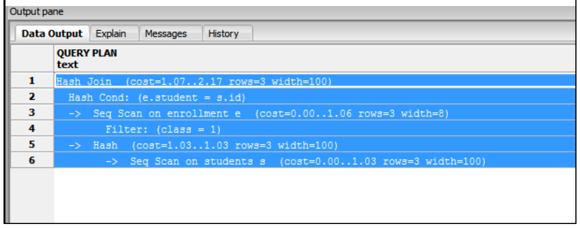
Cost: Assuming *R* is clustered and non-sorted and the index on *S* is clustered on *B* then

$B(R)+T(R)B(S)/V(S,B)$ + some more for reading index

Question: What is the cost if *R* is sorted?

Reading the plan that was chosen by the database (EXPLAIN)

EXPLAIN SELECT s.pid, s.first_name, s.last_name, e.credits
FROM students s, enrollment e
WHERE s.id = e.student
AND e.class = 1;



Notes on physical operators of Postgres and other databases

σ_c R turns into single operator

- Sequential Scan with filter c

Seq Scan on R
Filter: (c)

- Index Scan

Index Scan using <index> on R
Index Cond: (c)

201

Steps of joins, aggregations broken into fine granularity operators

- No sort-merge: Separate sort and merge
- Hash join has separate operation creating hash table and separate operation doing the looping

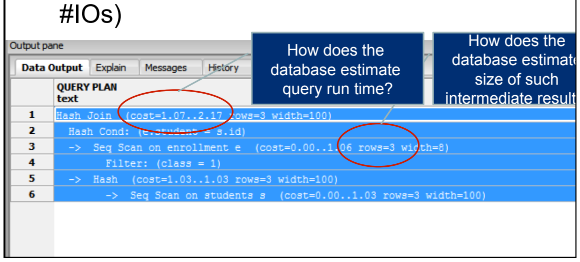
Sorting

- Sorting may be accomplished using index
  - Rarely wins 2-phase sort if table is not clustered and is much bigger than memory

- Estimating cost of query plan

(1) Estimating <u>size</u> of results

(2) Estimating run time (often reduces to #IOs)



Estimating result size

- 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

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 S(R) = 37
V(R,A) = 3 V(R,C) = 5
V(R,B) = 1 V(R,D) = 4

Size estimates for W = R1 x R2

T(W) = T(R1) × T(R2)

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

Size estimate for $W = \sigma_{Z=val}(R)$

S(W) = S(R)

T(W) = ?

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

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

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

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

T(W) = ?

- Solution # 1:
  T(W) = T(R)/2
- Solution # 2:
  T(W) = T(R)/3

- Solution # 3: Estimate values in range

Example R

	Z

Min=1 ↕ Max=20

V(R,Z)=10

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

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

T(W) = f × T(R)

Equivalently:

f×V(R,Z) = fraction of distinct values

$T(W) = [f \times V(Z,R)] \times \frac{T(R)}{V(Z,R)} = f \times T(R)$

<u>Size estimate</u> for W = R1 ⋈ R2

Let x = attributes of R1

y = attributes of R2

Case 1

$X \cap Y = \varnothing$

Same as R1 x R2

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

R1 | A | B | C R2 | A | D

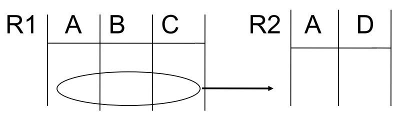
Assumption:

$\Pi_A R1 \subseteq \Pi_A R2 \Rightarrow$ Every A value in R1 is in R2 (typically A of R1 is foreign key of the primary key of A of R2)

$\Pi_A R2 \subseteq \Pi_A R1 \Rightarrow$ Every A value in R2 is in R1

"containment of value sets" (justified by primary key – foreign key relationship)

Computing T(W) when A of R1 is the foreign key $\Pi_A R1 \subseteq \Pi_A R2$

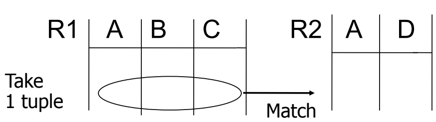


1 tuple of R1 matches with exactly 1 tuple of R2

so $T(W) = T(R1)$

Another way to approach when

$\Pi_A R1 \subseteq \Pi_A R2$



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

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

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

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

[A is common attribute]

<u>In general</u>   W = R1 ⋈ R2

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

Combining estimates on subexpressions:
Value preservation

R(A, C)
$T(R) = 10^3$
$V(A, R) = 10^3$
$V(C, R) = 10^2$

S(A, B)
$T(S) = 10^2$
V(A, S) = 50

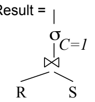
T(R ⋈ S) =
$T(R) \times T(S) / \max(V(A,R), V(A, S)) = 10^2$

$V(C, R \bowtie S) = 10^2$ ← (Big) assumption: Value preservation of C

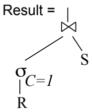
$T(Result) = T(R \bowtie S) / V(C, R \bowtie S) = 1$

Value preservation may have to be pushed to a weird assumption (but there's logic behind it!)

Result =



Result =



Ideally, the size estimation should not depend on which of the two equivalent formulas for Result one uses. However, to achieve this we may need to push the value preservation assumption to artificial intermediate estimates…

R(A, C)
$T(R) = 10^3$
$V(A, R) = 10^3$
$V(C, R) = 10^2$

S(A, B)
$T(S) = 10^2$
$V(A, S) = 50$

$T(R \bowtie S) = 10^2$

$V(C, R \bowtie S) = 10^2$

T(Result) = 1

$T(\sigma_{c=1}R) = T(R) / V(C, R) = 10$
$V(A, \sigma_{c=1}R) = \mathbf{10^3}$
T(Result) =
$T(\sigma_{c=1}R) \times T(S) / \max(V(A, \sigma_{c=1}R), V(A, S)) = 1$

We had to extend value preservation to the weird assumption that attribute A has more values than the number of tuples in R. In this way the number of S tuples matching an R tuple stays steady

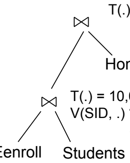
Value preservation of join attribute

CSEenroll(EID, SID, …)
T(CSEenroll) = 10,000
V(SID, CSEenroll) = 1,000

Students(SID, …)
T(Students) = 20,000
V(SID, Students) = 20,000

Honors (HID, SID, …)
T(Honors) = 5,000
V(SID, Honors) = 500

$T(\text{CSEenroll(EID, SID, …)} \bowtie \text{Students(SID, …)} \bowtie \text{Honors (HID, SID, …)}) = ?$



T(.) = **10,000 x 5,000 / max(500, <u>20,000</u>) = 2,500 CORRECT**
10,000 x 5,000 / max(500, <u>1,000</u>) = 50,000 WRONG

T(.) = 10,000
V(SID, .) ?= 1,000 (preservation of SIDs in CSEenroll)
or 20,000 (preservation of SIDs in Students) ?

If in doubt, think in terms of probabilities and matching records

- A SID of Student appears in CSEEnroll with probability 1000/20000
  - i.e., 5% of students are enrolled in CSE
- A SID of Student appears in Honors with probability 500/20000
  - i.e., 2.5% of students are honors students

=> An SID of Student appears in the join result with probability 5% x 2.5%

- On the average, each SID of CSEEnroll appears in 10,000/1,000 tuples
  - i.e., each CSE-enrolled student has 10 enrollments
- On the average, each SID of Honors appears in 5,000/500 tuples
  - i.e., each honors' student has 10 honors

⇒Each Student SID that is in both Honors and CSEEnroll is in 10x10 result tuples
⇒ T(result) = 20,000 x 5% x 2.5% x 10 x 10 = 2,500 tuples

CSEenroll(EID, SID, …)
T(Students) = 10,000
V(SID, Students) = 1,000

Students(SID, …)
T(Students) = 20,000
V(SID, Students) = 20,000

Honors (HID, SID, …)
T(Students) = 5,000
V(SID, Students) = 500

$T(\text{CSEenroll(EID, SID, …)} \bowtie \text{Students(SID, …)} \bowtie \text{Honors (HID, SID, …)}) = ?$

Plan Enumeration: Yet another source of suboptimalities

Not all possible equivalent plans are generated

- Possible rewritings may not happen
- Join sequences of n tables lead to #plans that is exponential in n
  - Eg, Postgres comes with a default exhaustive search for up to 12 joins

Morale: The plan you have in mind have not been considered

Arranging the Join Order: the Wong-Youssefi algorithm (INGRES)

Sample TPC-H Schema

```
Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
Lineitem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, SName)
```

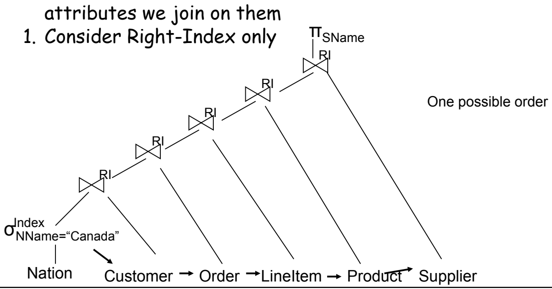
```
SELECT SName
FROM Nation, Customer, Order, LineItem, Product, Supplier
WHERE Nation.NationKey = Cuctomer.NationKey
      AND Customer.CustKey = Order.CustKey
      AND Order.OrderKey=LineItem.OrderKey
      AND LineItem.PartKey= Product.Partkey
      AND Product.Suppkey = Supplier.SuppKey
      AND NName = "Canada"
```

Find the names of suppliers that sell a product that appears in a line item of an order made by a customer who is in Canada

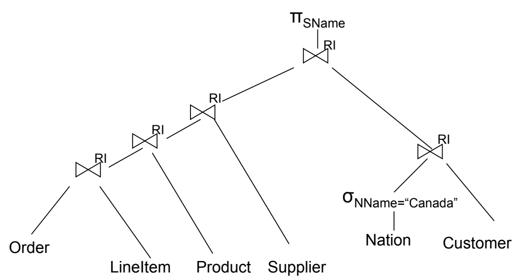
Challenges with Large Natural Join Expressions

For simplicity, assume that in the query

1. All joins are natural
2. whenever two tables of the FROM clause have common attributes we join on them
1. Consider Right-Index only



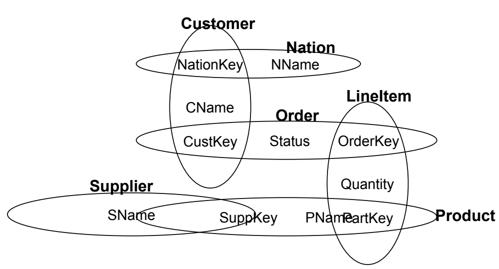
Multiple Possible Orders



Wong-Yussefi algorithm assumptions and objectives

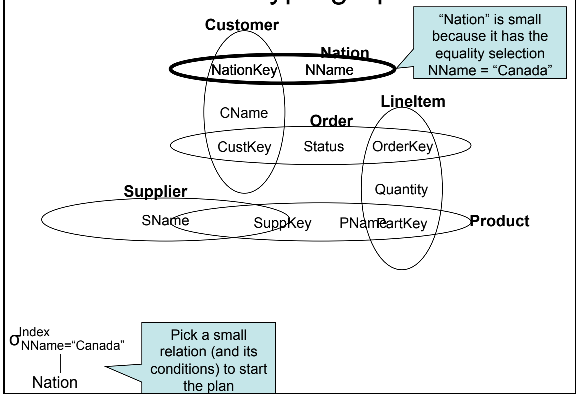
- Assumption 1 (weak): Indexes on all join attributes (keys and foreign keys)
- Assumption 2 (strong): At least one selection creates a *small* relation
  - A join with a small relation results in a small relation
- Objective: Create sequence of index-based joins such that all intermediate results are small

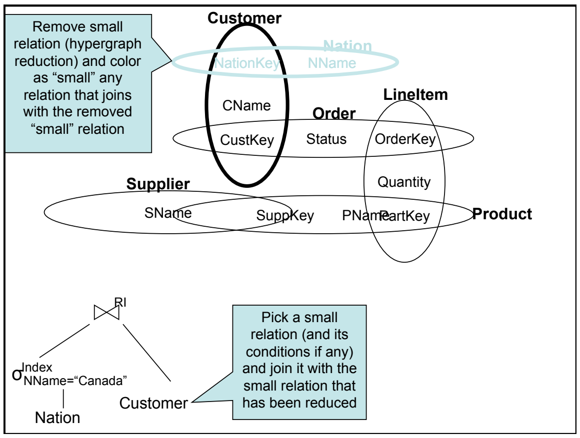
Hypergraphs



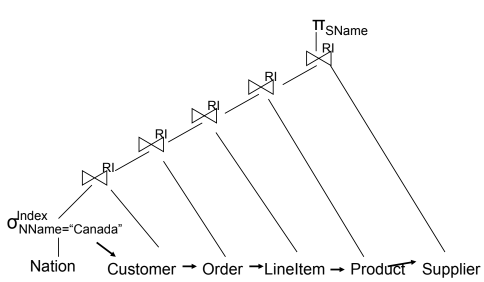
- relation hyperedges
  - two hyperedges for same relation are possible
- each node is an attribute
- can extend for non-natural equality joins by merging nodes

Small Relations/Hypergraph Reduction





After a bunch of steps…

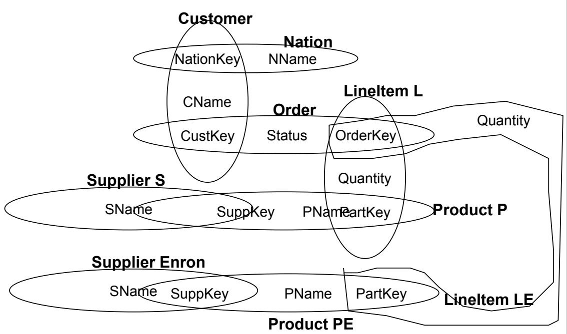


Multiple Instances of Each Relation

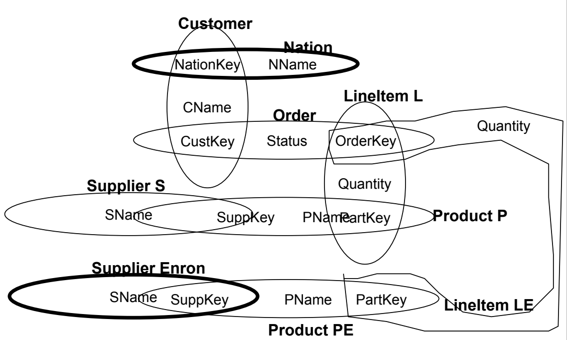
```
SELECT S.SName
FROM Nation, Customer, Order, LineItem L, Product P, Supplier S,
              LineItem LE, Product PE, Supplier Enron
WHERE Nation.NationKey = Cuctomer.NationKey
       AND Customer.CustKey = Order.CustKey
       AND Order.OrderKey=L.OrderKey
       AND L.PartKey= P.Partkey
       AND P.Suppkey = S.SuppKey
       AND Order.OrderKey=LE.OrderKey
       AND LE.PartKey= PE.Partkey
       AND PE.Suppkey = Enron.SuppKey
       AND Enron.Sname = "Enron"
       AND NName = "Cayman"
```

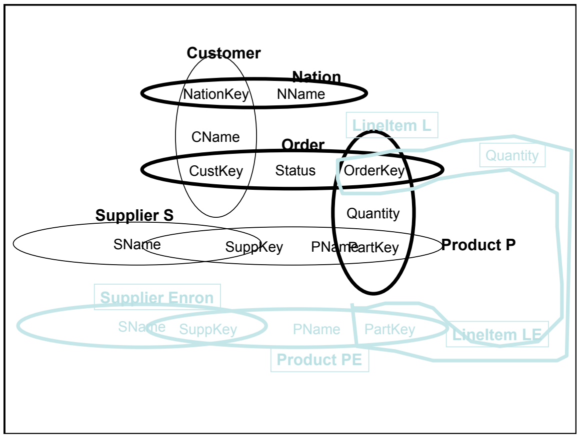
Find the names of suppliers whose products appear in an order made by a customer who is in Cayman Islands and an Enron product appears in the same order

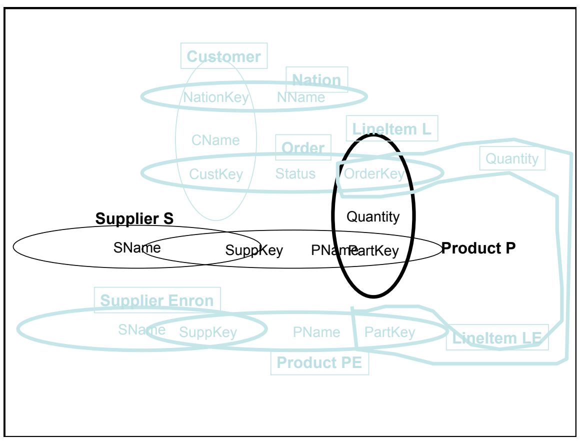
Multiple Instances of Each Relation

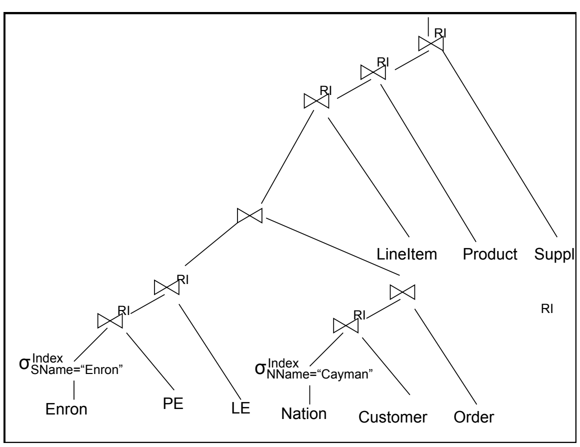


Multiple choices are possible




Customer
Nation
NationKey
NName
CName
Order
CustKey
Status
OrderKey
LineItem L
Quantity
Supplier S
SName
SuppKey
PName
PartKey
Product P
Supplier Enron
Product PE
LineItem LE


Customer
Nation
NationKey
NName
CName
Order
CustKey
Status
OrderKey
LineItem L
Quantity
Supplier S
SName
SuppKey
PName
PartKey
Product P
Supplier Enron
Product PE
LineItem LE


RI
LineItem
Product
Suppl
σIndex SName="Enron"
Enron
PE
LE
σIndex NName="Cayman"
Nation
Customer
Order

The basic dynamic programming approach to enumerating plans

for each sub-expression

$op(e_1\ e_2 \ldots e_n)$ of a logical plan

- (recursively) compute the best plan and cost for each subexpression e_i
- for each physical operator op^p implementing op
  - evaluate the cost of computing op using op^p and the best plan for each subexpression e_i
  - (for faster search) memo the best op^p

Local suboptimality of basic approach and the Selinger improvement

- Basic dynamic programming may lead to (globally) suboptimal solutions
- Reason: A suboptimal plan for e_1 may lead to the optimal plan for $op(e_1\ e_2 \ldots e_n)$
  - Eg, consider $e_1 \bowtie_A e_2$ and
  - assume that the optimal computation of e_1 produces unsorted result
  - Optimal $\bowtie$ is via sort-merge join on A
  - It could have paid off to consider the suboptimal computation of e_1 that produces result sorted on A
- Selinger improvement: memo also any plan (that computes a subexpression) and produces an order that may be of use to ancestor operators

Using dynamic programming to optimize a join expression

- Goal: Decide the join order and join methods
- Initiate with n-ary join $\bowtie_c(e_1\ e_2 \ldots e_n)$, where c involves only join conditions
- Bottom up: consider 2-way non-trivial joins, then 3-way non-trivial joins etc
  - "non trivial" -> no cartesian product

Summary

We learned

- how a database processes a query
- how to read the plan the database chose
  - Including size and cost estimates

Back to action:

- Choosing Indices, with our knowledge of cost with and without indices
- What if the database cannot find the best plan?