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.)
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

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 \( R.A = \text{<provided value>} \)
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.

\[
\text{SELECT *}
\text{FROM R}
\text{WHERE R.A = ?}
\]

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 \( O(n/m) \)

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
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 \( x \).

Indexing will help any query step when the problem is...

**Given condition on attribute find qualified records**

\[
\text{Attr} = \text{value} \quad ? \quad \text{Qualified records}
\]

Condition may also be

- \( \text{Attr} > \text{value} \)
- \( \text{Attr} \geq \text{value} \)
Indexing

- Data Structures used for quickly locating tuples that meet a specific type of condition
  - *Equality* condition: find Movie tuples where Director = \(X\)
  - Other conditions possible, e.g., *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

```sql
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
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
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

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
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

Clustered/sequential access-based algorithms for disk became relatively better

Moore’s Law: Same Phenomenon Applies to RAM

Algorithms that access memory sequentially have better constant factors than algorithms that access randomly
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

**Phase 1, round 1**
2-Phase Merge Sort

Phase 1, round 2
Phase 2 continues until no more records

In practice, probably many more Phase 1 rounds and many respective output files
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 $\#files < M/B$
  =>$\Rightarrow$ there can be up to $M/B$ Phase 1 rounds
- Each round can process up to $M$ bytes of input data
  =>$\Rightarrow$ 2-Phase Merge Sort can sort $M^2/B$ 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

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>number</th>
<th>date_code</th>
<th>start_time</th>
<th>end_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web</td>
<td>CSE135</td>
<td>TuTh</td>
<td>2:00</td>
<td>3:20</td>
</tr>
<tr>
<td>2</td>
<td>Databases</td>
<td>CSE132A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
</tr>
<tr>
<td>4</td>
<td>VLSI</td>
<td>CSE121</td>
<td>F</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Enrollment

<table>
<thead>
<tr>
<th>id</th>
<th>class</th>
<th>student</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Students

<table>
<thead>
<tr>
<th>id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8888888</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>1111111</td>
<td>Mary</td>
<td>Doe</td>
</tr>
<tr>
<td>3</td>
<td>2222222</td>
<td>null</td>
<td>Chen</td>
</tr>
</tbody>
</table>

Pack maximum #records per block

```
1 Web CSE135 TuTh 2:00 3:20 2 Databases CSE132A TuTh 3:30 4:50 4 VLSI CSE121 F null
```

“pack” each block with maximum # records
Utilize overflow blocks for insertions with “out of order” primary keys

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>number</th>
<th>date_code</th>
<th>start_time</th>
<th>end_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web</td>
<td>CSE135</td>
<td>TuTh</td>
<td>2:00</td>
<td>3:20</td>
</tr>
<tr>
<td>2</td>
<td>Databases</td>
<td>CSE132A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
</tr>
<tr>
<td>3</td>
<td>PL</td>
<td>CSE130</td>
<td>TuTh</td>
<td>9:00</td>
<td>9:50</td>
</tr>
<tr>
<td>4</td>
<td>VLSI</td>
<td>CSE121</td>
<td>F</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

... 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
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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>120</td>
<td>140</td>
<td>150</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequential File

Dense Primary Index

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

Sparse Primary Index

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

+ less index space

more + and - in a while
Sparse vs. Dense Tradeoff

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

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

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
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
Secondary indexes

File not sorted on secondary search key

• Sparse index

does not make sense!
• Dense index

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

Duplicate values & secondary indexes
Duplicate values & secondary indexes

one option...

Problem: excess overhead!
• disk space
• search time

Duplicate values & secondary indexes

another option: lists of pointers

Problem: variable size records in index!
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

Duplicate values & secondary indexes

buckets
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

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

Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's
This idea used in

text information retrieval

Documents

...the cat is fat ...

...my cat and my dog like each other...

..Fido the dog ...

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
Outline:

- Conventional indexes
- B-Trees \( \Rightarrow \) NEXT
- Hashing schemes
• **NEXT**: Another type of index
  – Give up on sequentiality of index
  – Try to get “balance”
Sample non-leaf

Sample leaf node:

From non-leaf node

to next leaf in sequence
In textbook’s notation \( n=3 \)

Leaf:

Non-leaf:

Size of nodes:

\[
\begin{align*}
\text{n+1 pointers} \\
\text{n keys} &\quad \text{(fixed)}
\end{align*}
\]
Non-root nodes have to be at least half-full

- Use at least

  Non-leaf: \(\lceil \frac{n+1}{2} \rceil\) pointers

  Leaf: \(\lfloor \frac{n+1}{2} \rfloor\) pointers to data
**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”

---

**Number of pointers/keys for B+tree**

<table>
<thead>
<tr>
<th></th>
<th>Max ptrs</th>
<th>Max keys</th>
<th>Min ptrs</th>
<th>Min keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf (non-root)</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>( \lceil (n+1)/2 \rceil )</td>
<td>( \lceil (n+1)/2 \rceil - 1 )</td>
</tr>
<tr>
<td>Leaf (non-root)</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>( \lfloor (n+1)/2 \rfloor )</td>
<td>( \lfloor (n+1)/2 \rfloor )</td>
</tr>
<tr>
<td>Root</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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

\[
\begin{array}{c}
\text{3} \\
\text{5} \\
\text{11} \\
\end{array}
\quad\quad
\begin{array}{c}
\text{30} \\
\text{31} \\
\text{32} \\
\end{array}
\quad\quad
\begin{array}{c}
\text{30} \\
\text{100} \\
\end{array}
\]
(a) Insert key = 7

(c) Insert key = 160
(d) New root, insert 45

Deletion from B+tree

(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf
(b) Coalesce with sibling
  – Delete 50

(c) Redistribute keys
  – Delete 50
(d) Non-leaf coalesce
   – Delete 37

B+tree deletions in practice

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

Hardware+ indexing problem:
For B+tree, how large should $n$ be?

\[ n \text{ is number of keys / node} \]
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 $12 + 0.003n$
  - 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:

$f(n) = \text{time to find a record}$

![Graph showing $f(n)$ versus $n$ with $n_{opt}$ as the optimum page size.]
FIND $n_{opt}$ by $f'(n) = 0$

Answer should be $n_{opt} = \text{"few hundred"}$

What happens to $n_{opt}$ as

- Disk gets faster?
- CPU get faster?

Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B+ trees
- Hashing schemes $\rightarrow$ Next
- Bitmap indices
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 = ‘\( x_1 \ x_2 \ldots \ x_n \)’ \( n \) byte character string
- Have \( b \) buckets
- \( h: \) add \( x_1 + x_2 + \ldots + x_n \)
  - compute sum modulo \( b \)
This may not be best function ... Read Knuth Vol. 3 if you really need to select a good function.

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

Within a bucket:

• Do we keep keys sorted?
  • Yes, if CPU time critical & Inserts/Deletes not too frequent
Next: example to illustrate inserts, overflows, deletes

EXAMPLE 2 records/bucket

INSERT:

- $h(a) = 1$
- $h(b) = 2$
- $h(c) = 1$
- $h(d) = 0$
- $h(e) = 1$
Rule of thumb:

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

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

- If < 50%, wasting space
- If > 80%, overflows significant depends on how good hash function is & on # keys/bucket
How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

Extensible hashing: two ideas

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

$h(K) \rightarrow \begin{array}{c}
00110101
\end{array}$

use $i \rightarrow$ grows over time....
(b) Use directory

\[ h(K)[0-i] \to \text{bucket} \]

Example: \( h(k) \) is 4 bits; 2 keys/bucket

\[ i = 1 \]

“slide” conventions:
slide shows \( h(k) \), while actual directory has key+pointer
Example: \( h(k) \) is 4 bits; 2 keys/bucket

Insert 1010

New directory

Example continued

Insert: 0111 0000
Example continued

Extensible hashing: deletion

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

- Run thru insert example in reverse!

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)
Linear hashing

• Another dynamic hashing scheme

Two ideas:
(a) Use $i$ low order bits of hash

(b) File grows linearly

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

$0101$

$m = 01$ (max used block)

Rule

If $h(k)[i] \leq m$, then
look at bucket $h(k)[i]$
else, look at bucket $h(k)[i] - 2^i - 1$
Example  \( b=4 \) bits,  \( i=2, \)  2 keys/bucket

- insert 0101

Future growth buckets

\[ m = 01 \] (max used block)

Example Continued: How to grow beyond this?

\[ i = 2^3 \]

Future growth buckets

\[ m = 11 \] (max used block)
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 \( U > \text{threshold} \) then increase \( m \)
  
  (and \( i \), when \( m \) reaches \( 2^i \))

Linear Hashing

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

- No indirection like extensible hashing

- Can still have overflow chains
Example: BAD CASE

Very full

Very empty

Need to move

Would waste space...

---

Summary

Hashing

- How it works
  - Dynamic hashing
    - Extensible
    - Linear
Next:

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

Hashing good for probes given key
e.g.,

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

```sql
SELECT
FROM R
WHERE R.A > 5
```

Index definition in SQL

• Create **index name** on **rel (attr)**
• Create **unique index name** on **rel (attr)**
  
  $\downarrow$
  
  defines candidate key

• **Drop INDEX** name
CANNOT SPECIFY TYPE OF INDEX
(e.g. B-tree, Hashing, ...)

OR PARAMETERS
(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

Note

ATTRIBUTE LIST ⇒ MULTIKEY INDEX
(next)
e.g., CREATE INDEX foo ON R(A,B,C)
Motivation: Find records where
DEPT = “Toy” AND SAL > 50k

Strategy I:

• Use one index, say Dept.
• Get all Dept = “Toy” records and check their salary
Strategy II:

- Use 2 Indexes; Manipulate Pointers

Toy $\rightarrow$ I$_1$ $\rightarrow$ I$_2$ $\rightarrow$ I$_3$ $\rightarrow$ Sal $\leftarrow$ I$_3$

Strategy III:

- Multiple Key Index

One idea:
For which queries is this index good?

- Find RECs Dept = “Sales” \( \land \) SAL = 20k
- Find RECs Dept = “Sales” \( \land \) SAL \( \geq \) 20k
- Find RECs Dept = “Sales”
- Find RECs SAL = 20k
Interesting application:

- Geographic Data

DATA:
- \(<X_1, Y_1, \text{Attributes}>\)
- \(<X_2, Y_2, \text{Attributes}>\)
- \(\vdots\)

Queries:

- What city is at \(<X_i, Y_i>\)?
- What is within 5 miles from \(<X_i, Y_i>\)?
- Which is closest point to \(<X_i, Y_i>\)?
Queries

- Find points with $Y_i > 20$
- Find points with $X_i < 5$
- Find points “close” to $i = <12,38>$
- Find points “close” to $b = <7,24>$
• Many types of geographic index structures have been suggested
  • Quad Trees
  • R Trees

Outline/summary

• Conventional Indexes
  • Sparse vs. dense
  • Primary vs. secondary

• B+ trees
• Hashing schemes
• Bitmap indices --> Next
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!

+ 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
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])

Toy: 00011010

- First run says: The first ace appears after 3 zeros
- Second run says: The 2nd ace appears immediately after the 1st
- Third run says: The 3rd ace appears after 1 zero after the 2nd

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
0000101, 10000001, 01000100, 00010001
Bit-aligned $2n \log m$ Compression (simple version)

Toys: 00011010

First run says:
The first ace appears after 3 zeros

Second run says:
The 2nd ace appears immediately after the 1st

Third run says:
The 3rd ace appears after 1 zero after the 2nd

1011 00 0 1

10 says: The binary encoding of the first number needs 1+1 digits.
11 says: The first number is 3

2nlog m compression

• Example
• Pens: 01000001
• Sequence [1,5]
• Encoding: 01 11 0 10 0 1
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)

Summing Up...

We discussed how the database stores data + basic algorithms

- Sorting
- Indexing
How are they used in query processing?
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
• 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)

<table>
<thead>
<tr>
<th>RxS</th>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bingo! Got one...
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

"FLY" and "SCAN" are the defaults
Another idea:

Plan II

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

<table>
<thead>
<tr>
<th>R</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>
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 \( \pi_A \), 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 the above as a rewriting rule
  - **Theta Join**
    - arbitrary condition involving multiple attributes

Grouping and Aggregation

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

Find the average salary for each department

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Toys</td>
<td>45</td>
</tr>
<tr>
<td>Nick</td>
<td>PCs</td>
<td>50</td>
</tr>
<tr>
<td>Jim</td>
<td>Toys</td>
<td>35</td>
</tr>
<tr>
<td>Jack</td>
<td>PCs</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>Joe</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Jim</td>
<td>35</td>
</tr>
<tr>
<td>PCs</td>
<td>Nick</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Jack</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>AvgSal</th>
<th>SalaryExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>PCs</td>
<td>45</td>
<td>90</td>
</tr>
</tbody>
</table>
Sorting and Lists

- SQL and algebra results are ordered
- Could be non-deterministic or dictated by SQL ORDER BY, algebra $\tau$
- $\tau_{OrderByList}$
- A result of an algebraic expression $o(exp)$ is ordered if
  - If $o$ is a $\tau$
  - 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

**Commutativity**

<table>
<thead>
<tr>
<th>Cartesian Product</th>
<th>Natural Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R \times S$ $\rightarrow$ $S \times R$</td>
<td>$R \bowtie S$ $\rightarrow$ $S \bowtie R$</td>
</tr>
</tbody>
</table>

**Associativity**

| $R \times S \times T$ | $R \bowtie S \bowtie T$ |

**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)

<table>
<thead>
<tr>
<th>Union</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R \cup S$ $\rightarrow$ $S \cup R$</td>
<td>$R \cap S$ $\rightarrow$ $S \cap R$</td>
</tr>
</tbody>
</table>

**Commutativity**

| $R \cup S \cup T$ | $R \cap S \cap T$ |

**Associativity**

| $R \cup (S \cup T)$ $\rightarrow$ $(R \cup S) \cup T$ | $R \cap (S \cap T)$ $\rightarrow$ $(R \cap S) \cap T$ |

**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

| \sigma_{\neg \text{cond2}} | \sigma_{\text{cond1}} AND \neg \text{cond2} | R | R' |
Pushing the Selection Thru Binary Operators: Union and Difference

Exercise: Do the rule for intersection

Pushing Selection thru Cartesian Product and Join

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only

The right direction requires that all the attributes used by \( \text{cond} \) appear in both \( R \) and \( S \)

Exercise: Do the rule for theta join
**Rules:** $\pi, \sigma$ combined

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

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

---

**Pushing Simple Projections Thru Binary Operators**

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

![Diagram](image)

**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: \( \sigma + \bowtie \) combined

More Rules can be Derived:

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

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

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

\( p \) only at \( R \), \( q \) only at \( S \), \( m \) at both \( R \) and \( S \)

\(--\) Derivation for first one:

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

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

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

\([\sigma_p (R)] \bowtie [\sigma_q (S)]\)
Which are always “good” transformations?

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

**In textbook:** more transformations

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

- No transformation is always 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

```
<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>37</td>
</tr>
</tbody>
</table>
```
Clustering can radically change cost

- Clustered relation
  
  $$\begin{array}{cccc}
  R1 & R2 & R3 & R4 \\
  R5 & R5 & R7 & R8 \\
  \end{array}$$
  
- 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

```java
class project
{
    open()
    { return child.open(); }

    getNext()
    { return child.getNext(); }
}
```
Example \( R_1 \bowtie R_2 \) 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

  \[
  \text{for each } r \in R_1 \text{ do } \\
  \quad \text{for each } s \in R_2 \text{ do } \\
  \quad \quad \text{if } r.C = s.C \text{ then output } r,s \text{ pair}
  \]

Also called “nested loop join” in some databases (eg Postgres)
• Join with index (Conceptually)
  – alike iteration join but right relation
    accessed with index

For each $r \in R1$ do
  
  \[ X \leftarrow \text{index} \left( R2, C, r.C \right) \]
  
  for each $s \in X$ do
    
    output $r,s$ pair

\textbf{Assume R2.C index}

\textbf{Note:} \( X \leftarrow \text{index}\left( \text{rel, attr, value} \right) \)
  
  then \( X \) = set of rel tuples with attr = value

• Merge join (conceptually)
  
  (1) if $R1$ and $R2$ not sorted, sort them
  (2) $i \leftarrow 1$; $j \leftarrow 1$
  
  While ($i \leq T(R1)) \land (j \leq T(R2))$ do
    
    if $R1\{i\}.C = R2\{j\}.C$ then outputTuples

    else if $R1\{i\}.C > R2\{j\}.C$ then $j \leftarrow j+1$

    else if $R1\{i\}.C < R2\{j\}.C$ then $i \leftarrow i+1$
Procedure Output-Tuples

While \((R1\{i\}.C = R2\{j\}.C) \land (i \leq T(R1))\) do

\[jj \leftarrow j;\]

while \((R1\{i\}.C = R2\{jj\}.C) \land (jj \leq T(R2))\) do

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

\[jj \leftarrow jj+1 \]

\[i \leftarrow i+1 \]

Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{i}.C</th>
<th>R2{j}.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
• 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

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>Even</th>
<th>Odd:</th>
<th>Buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>2 4 8</td>
<td>3 5 9</td>
<td>4 12 8 14</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

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 \( B_r \) of \( R \) do
    store tuples of \( B_r \) in main memory
for each each block \( B_s \) of \( S \) do
    for each tuple \( s \) of \( B_s \)
        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 \( B_r \) of \( R \) do
    store tuples of \( B_r \) in main memory
    hash buckets \( G_1, \ldots, G_n \)
for each each block \( B_s \) of \( S \) do
    for each tuple \( s \) of \( B_s \)
        find \( h=\text{hash}(s) \)
        join \( s \) with matching tuples in \( G_h \)
Generalization of Nested-Loops

for each chunk of $M-1$ blocks $B_r$ of $R$ do
    store tuples of $B_r$ in main memory
    for each each block $B_s$ of $S$ do
        for each tuple $s$ of $B_s$
            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

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.

<table>
<thead>
<tr>
<th>$R$</th>
<th>$C$</th>
<th>$RA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$r_1$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$r_2$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$r_3$</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>$r_{20}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>$C$</th>
<th>$SA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$s_1$</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$s_5$</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>$s_{16}$</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>$s_{20}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R$</th>
<th>3 7 8 10 11 13 14 16 17 18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 5 6 9 12 15 19 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$S$</th>
<th>1  3  5  17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2  4  16 18 19 20</td>
</tr>
</tbody>
</table>
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 \( i=1\ldots M-1 \)
  – 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 \( \{s_1 ,s_2 ,\ldots ,s_n \} \) of \( S \) with \( B=b \)
  output \( \{rs_1 ,rs_2 ,\ldots ,rs_n \} \)

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
$\sigma_c R$ turns into single operator

- Sequential Scan with filter $c$
  
  Seq Scan on $R$
  Filter: $(c)$

- Index Scan
  
  Index Scan using $<\text{index}>$ on $R$
  Index Cond: $(c)$

---

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 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 size of results
  (2) Estimating run time (often reduces to #IOs)

Both estimates can go very wrong!
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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>20</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>30</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>40</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>1</td>
<td>50</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

$T(R) = 5 \quad S(R) = 37$

$V(R, A) = 3 \quad V(R, C) = 5$

$V(R, B) = 1 \quad V(R, D) = 4$
Size estimates for $W = R_1 \times R_2$

$T(W) = T(R_1) \times T(R_2)$

$S(W) = S(R_1) + S(R_2)$

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

$S(W) = S(R)$

$T(W) = ?$
Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>V(R,A)</th>
<th>V(R,B)</th>
<th>V(R,C)</th>
<th>V(R,D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>1</td>
<td>10</td>
<td>a</td>
<td></td>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>20</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>30</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
<td>40</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bat</td>
<td>1</td>
<td>50</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$W = \sigma_{z,\text{val}(R)}$$

$$T(W) = \frac{T(R)}{V(R,Z)}$$

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

$$T(W) = ?$$

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

- **Solution # 2:**
  
  $$T(W) = T(R)/3$$
Solution # 3: Estimate values in range

Example

<table>
<thead>
<tr>
<th>R</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min=1 V(R,Z)=10</td>
</tr>
<tr>
<td></td>
<td>W= ( \sigma_{z \geq 15} (R) )</td>
</tr>
<tr>
<td></td>
<td>Max=20</td>
</tr>
</tbody>
</table>

\[
f = \frac{20-15+1}{20-1+1} = \frac{6}{20} = \frac{3}{10} \text{ (fraction of range)}
\]

\[
T(W) = f \times T(R)
\]

Equivalently:

\[
f \times V(R,Z) = \text{fraction of distinct values}
\]

\[
T(W) = \left[ f \times V(Z,R) \right] \times T(R) = \frac{f \times T(R)}{V(Z,R)}
\]
Size estimate for \( W = R_1 \bowtie R_2 \)

Let \( x = \) attributes of \( R_1 \)

\[ y = \) attributes of \( R_2 \]

\[ X \cap Y = \emptyset \]

Same as \( R_1 \times R_2 \)

Case 1

\[ W = R_1 \bowtie R_2 \]
\[ X \cap Y = A \]

Case 2

Assumption:

\[ \Pi_A R_1 \subseteq \Pi_A R_2 \Rightarrow \] Every \( A \) value in \( R_1 \) is in \( R_2 \)

(typically \( A \) of \( R_1 \) is foreign key of the primary key of \( A \) of \( R_2 \))

\[ \Pi_A R_2 \subseteq \Pi_A R_1 \Rightarrow \] Every \( A \) value in \( R_2 \) is in \( R_1 \)

“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 $T(W) = \frac{T(R2) \times T(R1)}{V(R2, A)}$
• $V(R_1,A) \leq V(R_2,A)$ \quad T(W) = \frac{T(R_2) T(R_1)}{V(R_2,A)}$

• $V(R_2,A) \leq V(R_1,A)$ \quad T(W) = \frac{T(R_2) T(R_1)}{V(R_1,A)}$

[A is common attribute]

\textbf{In general} \quad W = R_1 \Join R_2

\[
T(W) = \frac{T(R_2) T(R_1)}{\max\{ V(R_1,A), V(R_2,A) \}}
\]
Combining estimates on subexpressions:
Value preservation

\[
\text{Result} = \sigma_{C=1} R \bowtie S
\]

\[
\begin{align*}
R(A, C) & \quad S(A, B) \\
T(R) &= 10^3 \\
V(A, R) &= 10^3 \\
V(C, R) &= 10^2
\end{align*}
\]

\[
\begin{align*}
S(A, B) & \quad T(S) = 10^2 \\
V(A, S) &= 50
\end{align*}
\]

\[
T(R \bowtie S) = \frac{T(R) \times T(S)}{\max(V(A, R), V(A, S))} = 10^2
\]

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

(Big) assumption:
Value preservation of C

\[
T(\text{Result}) = \frac{T(R \bowtie S)}{V(C, R \bowtie S)} = 1
\]

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

\[
\begin{align*}
R(A, C) & \quad S(A, B) \\
T(R) &= 10^3 \\
V(A, R) &= 10^3 \\
V(C, R) &= 10^2
\end{align*}
\]

\[
\begin{align*}
S(A, B) & \quad T(S) = 10^2 \\
V(A, S) &= 50
\end{align*}
\]

\[
T(R) = 10^3
\]

\[
T(\sigma_{C=1} R) = T(R) / V(C, R) = 10
\]

\[
V(A, \sigma_{C=1} R) = 10^3
\]

\[
T(\text{Result}) = \frac{T(\sigma_{C=1} R) \times T(S)}{\max(V(A, \sigma_{C=1} R), V(A, S))} = 1
\]

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

\[
T(\text{Result}) = 1
\]

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…

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

\[ T(\text{CSEenroll}) = 10,000 \]
\[ V(\text{SID, CSEenroll}) = 1,000 \]

\[ T(\text{Students}) = 20,000 \]
\[ V(\text{SID, Students}) = 20,000 \]

\[ T(\text{Honors}) = 5,000 \]
\[ V(\text{SID, Honors}) = 500 \]

\[ T(\text{CSEenroll}(\text{EID}, \text{SID}, \ldots) \bowtie \text{Students}(\text{SID}, \ldots) \bowtie \text{Honors}(\text{HID}, \text{SID}, \ldots)) = ? \]

\[ T(\cdot) = \frac{10,000 \times 5,000}{\max(500, 20,000)} = 2,500 \text{ CORRECT} \]
\[ 10,000 \times 5,000 / \max(500, 1,000) = 50,000 \text{ WRONG} \]

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
  \[ \Rightarrow \text{An SID of Student appears in the join result with probability } 5\% \times 2.5\% \]
- 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
  \[ \Rightarrow \text{Each Student SID that is in both Honors and CSEEnroll is in 10x10 result tuples} \]
  \[ \Rightarrow T(\text{result}) = 20,000 \times 5\% \times 2.5\% \times 10 \times 10 = 2,500 \text{ tuples} \]
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, Lineltem, Product, Supplier
WHERE Nation.NationKey = Customer.NationKey
  AND Customer.CustKey = Order.CustKey
  AND Order.OrderKey = Lineltem.OrderKey
  AND Lineltem.PartKey = Product.Partkey
  AND Product.Suppkey = Supplier.SuppKey
  AND NName = “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

“Nation” is small because it has the equality selection NName = “Canada”

Pick a small relation (and its conditions) to start the plan

Remove small relation (hypergraph reduction) and color as “small” any relation that joins with the removed “small” relation

Pick a small relation (and its conditions if any) and join it with the small relation that has been reduced
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 = Customer.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

Customer
- NationKey
- CName
- CustKey
- Status
- OrderKey

Order
- Quantity
- OrderKey

Nation
- NName

Supplier S
- SName
- SuppKey
- PName

Product P
- PartKey

Supplier Enron
- SName
- SuppKey
- PName

Product PE
- PartKey

LinItem L
- Quantity

LinItem LE

Multiple choices are possible
The basic dynamic programming approach to enumerating plans

for each sub-expression \( \text{op}(e_1 e_2 \ldots e_n) \) of a logical plan

– (recursively) compute the best plan and cost for each subexpression \( e_i \)

– for each physical operator \( \text{op}^\rho \) implementing \( \text{op} \)
  • evaluate the cost of computing \( \text{op} \) using \( \text{op}^\rho \) and the best plan for each subexpression \( e_i \)
  • (for faster search) memo the best \( \text{op}^\rho \)
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 $e_1 \Join_A e_2$ and
  – assume that the optimal computation of $e_1$ produces unsorted result
  – Optimal $\Join_A$ 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 $\Join_C (e_1 e_2 \ldots e_n)$, 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?