Introduction to Data Management
Tuning and Indexes

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Goals for Today

- We gave a baseline for what join algorithms (and respective costs) were possible
- Use DB structures to expand optimization options
Recap – Plan Enumeration

SQL

```
SELECT *
FROM T, R, S
WHERE ...
```

Logical Plan

```
T R S
```

Eq. Logical Plans

```
T R S
R T S
S R T
```

Execution

```
100101010110
000101111010
100010101000
001010010100
```

Least Cost Plan

```
T R S
```

Physical Plans

```
T R S
R T S
S R T
```

RDBMS

SQL Logical Plan

Eq. Logical Plans

Execution

Least Cost Plan

Physical Plans
For this class we make a lot of assumptions

- **Disk-based storage**
  - HDD not SDD

- **Row-based storage**
  - Tuples are stored contiguously

- **IO cost only**
  - One disk access is ~100000x more expensive than one main memory access

- **Cold cache**
  - No data preloaded into main memory
Recap – Disk Storage

- Can only read 1 block per read operation
  - Usually 512B to 4kB

- Sequential disk reads are faster than random ones
  - Cost ~ 1-2% random scan = full sequential scan
Recap – Making Cost Estimations

- RDBMS keeps statistics about our tables
  - $B(R) = \# \text{ of blocks}$ in relation $R$
  - $T(R) = \# \text{ of tuples}$ in relation $R$
  - $V(\text{attr}, R) = \# \text{ of distinct values}$ of attr in $R$
Recap – Disk Storage

- Tables are stored as files
  - Heap file → Unsorted tuples
    - Nested-Loop Joins
      - Block-at-a-time Nested Loop Join (cost = \( B(R) + B(R) \times B(S) \))
      - Optimized Block-Nested-Loop Join (cost = \( B(R) + B(R)/N \times B(S) \))
    - Hash Join \( B(R)<M \) (all of relation fits in memory)
      cost = \( B(R) + B(S) \)
  
- **Sequential file** → Sorted tuples
Outline

- Index structures
- Index join cost estimation
- Database tuning
Indexing

- Indexes (for this class) can be assumed to be already loaded into memory
- An index does not have to contain all tuple data
  - Only key values are stored in the index
  - If an index contains all tuple data it is called a “covering index”
- Indexes are access points for tables
Index Structures

- **B+ Tree Index**
  - Clustered
  - Unclustered
- Hash Index
- R Tree
- R Tree
- Radix Tree
- Bloom Filter
- Hilbert Curves
- ...

November 8, 2019  Tuning  10
What is a B Tree?

“What, if anything, the B stands for has never been established.” – Wikipedia

- **Search tree** (like a binary search tree)
  - Nodes annotate max values
  - Large number of children per node

- Tree/node structure that is memory efficient

- Each entry of a node:
  - Left pointer to values less than entry
  - Right pointer to values greater than/equal to entry
What is a B Tree?

node

leaf node (base of tree)
What is a B Tree?

Find the value 40
What is a B Tree?

Find the value 40
What is a B Tree?

Find the value 40
What is a B Tree?

Find the value 40
How is a B+ Tree Different?

- Leaf nodes point to data
  - Data is searchable by **key** value annotated by the node labels
- Leaf nodes form a linked list
How is a B+ Tree Different?

Find the data associated with the key value 40
(same search process)
Clustered vs Unclustered Index

- An index is either **clustered** or **unclustered**, depending on how the actual data is sorted on disk.
A **clustered index** is one that has the **same** key ordering as what is on disk (one per table).
An unclustered index may exist without any ordering on disk (any number per table)

Sequential File with a different key or Heap File
Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85
Range queries can be fast!
- Filtering a value on a valid range is essentially looking up some portion of a B+ tree.

Find the data associated with the key values 40 to 85.
Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85
Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85
Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85
Benefits of B+ Trees

- Range queries can be fast!
  - Filtering a value on a valid range is essentially looking up some portion of a B+ tree

Find the data associated with the key values 40 to 85

Diagram showing how to find the data associated with the key values 40 to 85 in a B+ tree structure.
Estimating Amount of Data Read

- **Selectivity Factor** \( (X) \rightarrow \) Proportion of total data needed

- Assuming uniform distribution of data values on numeric attribute \( a \) in table \( R \), if the condition is:
  
  - \( a = c \) \( \rightarrow X \approx \frac{1}{V(a,R)} \) \( a = 4 \)
  
  - \( a < c \) \( \rightarrow X \approx \frac{c - \min(a,R)}{\max(a,R) - \min(a,R)} \) \( [1, 1, 2, 2, 3, 3, 4, 4, 5, 5] \) (10 entries, 5 unique values)
  
  - \( c_1 < a < c_2 \) \( \rightarrow X \approx \frac{c_2 - c_1}{\max(a,R) - \min(a,R)} \) \( X = 1 / 5 \)
  
  - \( \text{cond1 AND cond2} \) \( \rightarrow X \approx X_1 * X_2 \) \( 10 * 1/5 = 2 \)

- Disclaimer: More thorough selectivity estimation will use a histogram
Estimating Amount of Data Read

- **Selectivity Factor** \( (X) \rightarrow \) Proportion of total data needed

- Assuming uniform distribution of data values on numeric attribute \( a \) in table \( R \), if the condition is:
  - \( a=c \rightarrow X \approx \frac{1}{V(a,R)} \) \( a < 4 \)
  - \( a<c \rightarrow X \approx \frac{c-min(a,R)}{max(a,R)-min(a,R)} \[1, 1, 2, 2, 3, 3, 4, 4, 5, 5\]
  - \( c1<a<c2 \rightarrow X \approx \frac{c2-c1}{max(a,R)-min(a,R)} \) \( X = (4 - 1) / (5 - 1) \)
  - \( \text{cond1 AND cond2} \rightarrow X \approx X_1 \times X_2 \)

- **Disclaimer:** More thorough selectivity estimation will use a histogram
Estimating Amount of Data Read

- **Selectivity Factor** \((X) \rightarrow\) Proportion of total data needed

- Assuming uniform distribution of data values on numeric attribute \(a\) in table \(R\), if the condition is:
  - \(a=c\) \(\rightarrow\) \(X \cong \frac{1}{V(a,R)}\) \(\quad 2 < a < 4\)
  - \(a<c\) \(\rightarrow\) \(X \cong \frac{c-\text{min}(a,R)}{\text{max}(a,R)-\text{min}(a,R)}\) \(\quad \text{[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]}\)
  - \(c1<a<c2\) \(\rightarrow\) \(X \cong \frac{c2-c1}{\text{max}(a,R)-\text{min}(a,R)}\) \(\quad X = (4 - 2) / (5 - 1)\)
  - \(\text{cond1 AND cond2}\) \(\rightarrow\) \(X \cong X_1 \times X_2\)

- **Disclaimer:** More thorough selectivity estimation will use a histogram
Index-Based Selection

- For reference, a full sequential scan of data costs $B(R)$ IOs (block reads)
- Provided some condition to read data:
  - Full **sequential scan** $\rightarrow B(R)$
  - Scan on **clustered index** $\rightarrow X^*B(R)$
    - Able to read a contiguous chunk of the file
  - Scan on **unclustered index** $\rightarrow X^*T(R)$
    - Worst case would read a different block everytime
Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way”
Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way”
Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way.”
Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way”
Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way”.

Disk
Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. Without an index, finding a value must be done the “old fashioned way”.

Total cost is $B(R)$
Assume a block holds 4 tuples. I want tuples associated with values 40-85. With a clustered index, I start scanning blocks in the range they are at.
Assume a block holds 4 tuples. I want tuples associated with values 40-85. With a clustered index, I start scanning blocks in the range they are at.

Estimated cost is \( X \times B(R) \)
Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. With an unclustered index, I start scanning tuples wherever they occur.

Sequential File with a different key or Heap File
Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. With an unclustered index, I start scanning tuples wherever they occur.
Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. With an unclustered index, I start scanning tuples wherever they occur.

Sequential File with a different key or Heap File
Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. With an unclustered index, I start scanning tuples wherever they occur.
Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85. With an unclustered index, I start scanning tuples wherever they occur.

Estimated cost is \( X \times T(R) \)

Sequential File with a different key or Heap File
Index Expectations

- Using an index in the wrong scenario can lead to a slowdown!
- Common example: Full sequential scan vs unclustered index scan with high X value and/or small tuple size (large T(R):B(R) ratio)

Known:

- B(R) = 100
- T(R) = 10000

Consider a query with X=1/10

<table>
<thead>
<tr>
<th>Sequential scan</th>
<th>Index scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>= B(R)</td>
<td>= X*T(R)</td>
</tr>
<tr>
<td>= 100</td>
<td>= 1/10 * 10000</td>
</tr>
<tr>
<td></td>
<td>= 1000</td>
</tr>
</tbody>
</table>
Index Expectations

- Using an index in the wrong scenario can lead to a slowdown!

- Common example: Full sequential scan vs unclustered index scan with high X value and/or small tuple size (large T(R):B(R) ratio)

Having indexes doesn’t mean you will see a speedup!

Consider a query with X=1/10

**Known:**

\[ B(R) = 100 \]
\[ T(R) = 10000 \]

**Sequential scan**

\[ = B(R) \]
\[ = 100 \]

**Index scan**

\[ = X \cdot T(R) \]
\[ = 1/10 \cdot 10000 \]
\[ = 1000 \]
Index Expectations

- **Sequential disk reads are faster than random ones**
  - Cost $\sim$ 1-2% random scan = full sequential scan
  - Equivalent to selectivity $\sim$ .01-.02

![Graph showing the comparison between different data scan methods]
CREATE TABLE Users (  
id INT,  
age INT,  
score INT);

CREATE INDEX U_age ON Users(age)

CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
CREATE TABLE Users (  
id INT,  
age INT,  
score INT);

CREATE INDEX U_age ON Users(age)

CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
CREATE TABLE Users (    id INT,    age INT,    score INT);

CREATE INDEX U_age ON Users(age)

Unclustered by default

CREATE INDEX U_age_score ON Users(age, score)

Order specifies precedence in sorting

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
CREATE TABLE Users (  
id INT,  
age INT,  
score INT);

CREATE INDEX U_age ON Users(age)

CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)

Unclustered by default

Order specifies precedence in sorting

Reorders data on disk! (Fails if another clustered index exists)
CREATE INDEX U_age ON Users(age)

CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
Index-Based Equijoin

- Assume index exists on the join attribute \( a \) of \( S \)

```sql
SELECT *
FROM R, S
WHERE R.a = S.a
```

- Clustered Index Join
  - Perform a clustered index scan for each tuple of \( R \)
  - \( B(R) + T(R)(X*B(S)) = B(R) + T(R)(B(S)/V(a,S)) \)

- Unclustered Index Join
  - Perform an unclustered index scan for each tuple of \( R \)
  - \( B(R) + T(R)(X*T(S)) = B(R) + T(R)(T(S)/V(a,S)) \)
Index-Based Equijoin

- Assume index exists on the join attribute `a` of `S`

```sql
SELECT * FROM R, S WHERE R.a = S.a
```

### Clustered Index Join
- Perform a clustered index scan for each tuple of `R`
- \( B(R) + T(R)(X \ast B(S)) = B(R) + T(R)(B(S)/V(a,S)) \)

### Unclustered Index Join
- Perform an unclustered index scan for each tuple of `R`
- \( B(R) + T(R)(X \ast T(S)) = B(R) + T(R)(T(S)/V(a,S)) \)

Can’t scan per block of `R` since tuples in blocks don’t have the same attribute values.
Leveraging Indexes

- Often for applications, workloads can be well described
  - Flights application
    - Search method → query on city name values
  - Data visualization software (e.g. Tableau)
    - 2D plot → query on graph axis bounds

- **Create indexes to match expected query workload**
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

What indexes could we make on Users?

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

This range query would benefit from a clustered index on score

Only one can exist!

This range query would benefit from a clustered index on age
CREATE TABLE Users (  id INT PRIMARY KEY,  age INT,  score INT, ...);

Expecting 1000 exec/day
SELECT *
    FROM Users, Assets
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *
    FROM Users
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *
    FROM Users
WHERE Users.age > 21

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Things to consider:
• What is the expected result size for each query?
• Do either of these queries need to be returned ASAP?
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

What indexes could we make on Users?

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21

IDs are unique so an unclustered index would do fine.
Without more information, default to clustering on the index that will be used more (clustered index on score)
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21

Hack:  
• Create a covering index primarily keyed on score  
• Create a covering index primarily keyed on age
CREATE TABLE Users ( 
    id INT PRIMARY KEY, 
    age INT, 
    score INT, ...
);

What indexes could we make on Users?

Expecting 1000 exec/day

SELECT *
    FROM Users, Assets
WHERE Users.id = Assets.uid

Expecting 1000 exec/day

SELECT *
    FROM Users
WHERE Users.score > 95

Expecting 10 exec/day

SELECT *
    FROM Users
WHERE Users.age > 21

IDs are unique so an unclusted index would do fine.

Hack:
- Create a covering index primarily keyed on score
- Create a covering index primarily keyed on age

Essentially a sorted copy of the table. Fast but space inefficient and table updates are slow.
Choosing how to configure a database system is an interesting (i.e. hard) problem.

A database that is used by many people will often need one or more dedicated personnel to manage it (Database Administrator):

- Logical design (multi-team coordination)
- Physical design (hardware and system considerations)
- Permission management (visibility and security)
- Integration (company acquisitions and mergers)
- ...
Multiple Joins

- **Pipelined Execution**
  - Tuples are processed through the entire query plan
  - Fast
Pipelined Execution

- Iterator interface of RA operators (Volcano Iterator Model)
  - open() on every operator at start
  - close() on every operator at end
  - next() to get the next tuple from a child operator or input table
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
**Pipelined Execution Example**

- **Iterator interface of RA operators** *(Volcano Iterator Model)*

```
\sigma_{R.b=5} \bowtie S.a=T.a
```

```
\sigma_{R.b=5} \bowtie <\sigma_{R.b=5} \bowtie \bowtie R.a=S.a
```

```
open()
```

```
R \bowtie S
```

```
S \bowtie T
```

```
T
```

November 8, 2019
• Iterator interface of RA operators (*Volcano Iterator Model*)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)

\[ R \bowtie S \bowtie T \sigma_{R.b=5} \]

- Diagram:
  - R \bowtie S.a=T.a \rightarrow S \bowtie R.a=S.a \rightarrow R \sigma_{R.b=5} = 5 \rightarrow T \]
  - Open calls: open() for each operator.
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

```
\sigma_{R.b=5} \text{next()}
```

```
\bowtie_{R.a=S.a}
\bowtie_{S.a=T.a}
```

```
R \to S \to T
```
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)

```
\[ \sigma_{R.b=5} \quad \text{next()}
\]
```

```
\[ \bowtie_{S.a=T.a} \quad \text{next()}
\]
```

```
\[ \bowtie_{R.a=S.a} \quad \text{next()}
\]
```

```
\[ R \quad S \quad T
\]
```
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

next() implementation will depend on algorithm used (CSE 444)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

VOLCANO!!!

\[
\text{output} \\
\sigma_{R.b=5} \\
t_2 \\
\bowtie_{S.a=T.a} \\
t_1 \\
\bowtie_{R.a=S.a} \\
R \rightarrow \text{next()} \rightarrow \text{next()...} \\
S \rightarrow \text{next()} \rightarrow \text{next()...} \\
T \rightarrow \text{next()} \rightarrow \text{next()...}
\]