Speed Up Your Database Systems

CS307 Lab12, Spring 2019

ZIQIN WANG

Southern University of Science and Technology
11712310@mail.sustech.edu.cn

May 7, 2019

SUSTech Southern University of Science and Technology

Contents

Contents

What is index?

- How to search efficiently?

What is index?

- How to search efficiently?
  - Strategy: Divide and Conquer,
    e.g. *Binary Search*

What is index?

- How to search efficiently?
  - Strategy: Divide and Conquer,
    e.g. *Binary Search*
  - Assumption: Stored In Order

What is index?

- How to search efficiently?
  - Strategy: Divide and Conquer, e.g. *Binary Search*
  - Assumption: Stored In Order
- What if one want to search for ...
  - “The movie named ***Titanic***”
  - “**Chinese** movies released in **1999**”

What is index?

- How to search efficiently?
  - Strategy: Divide and Conquer,
    e.g. *Binary Search*
  - Assumption: Stored In Order
- What if one want to search for ...
  - "The movie named ***Titanic***"
    $\Longrightarrow$ `title`
  - "**Chinese** movies released in **1999**"
    $\Longrightarrow$ `country` & `year_released`

What is index?

- How to search efficiently?
  - Strategy: Divide and Conquer, e.g. *Binary Search*
  - Assumption: Stored In Order
- What if one want to search for ...
  - "The movie named ***Titanic***"
    $\Longrightarrow$ `title`
  - "**Chinese** movies released in **1999**"
    $\Longrightarrow$ `country` & `year_released`

But ...

Data are not stored in multiple orders

What is index?

Think about a Chinese Dictionary

What is index?

Think about a Chinese Dictionary

- Words are listed in alphabetical order

What is index?

Think about a Chinese Dictionary

- Words are listed in alphabetical order
- We still have
  - 音序检字法
  - 部首检字法
  - 笔顺检字法

What is index?

Think about a Chinese Dictionary

- Words are listed in alphabetical order
- We still have
    - 音序检字法
    - 部首检字法
    - 笔顺检字法
- How is this achieved?

What is index?

Concept

An *index* is a data structure which improves the **efficiency** of **retrieving** data with specific values from a database.

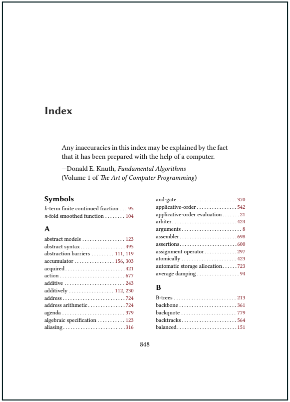
What is index?

Concept

An *index* is a data structure which improves the **efficiency** of **retrieving** data with specific values from a database.

Remark

- Indexes in databases is similar to indexes in books
  - Data are stored in *pages* (blocks)
  - Indexes locate a row by (filename, block #, offset)



Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

—Donald E. Knuth, *Fundamental Algorithms*
(Volume 1 of *The Art of Computer Programming*)

848

Figure: Index page of *Structure and Interpretation of Computer Programs*, 2nd ed., by H. Abelson, G. J. Sussman and J. Sussman, licensed under CC BY-SA 4.0.

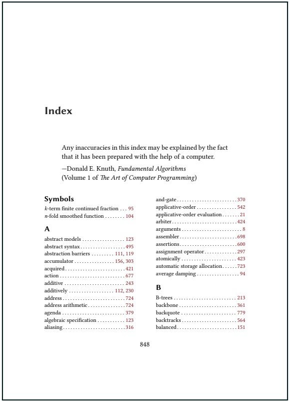
What is index?

Concept

An *index* is a data structure which improves the **efficiency** of **retrieving** data with specific values from a database.

Remark

- Indexes in databases is similar to indexes in books
  - Data are stored in *pages* (blocks)
  - Indexes locate a row by (filename, block #, offset)
- Indexing is beyond the scope of Relational Theory



Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

—Donald E. Knuth, *Fundamental Algorithms* (Volume 1 of *The Art of Computer Programming*)

848

Figure: Index page of *Structure and Interpretation of Computer Programs*, 2nd ed., by H. Abelson, G. J. Sussman and J. Sussman, licensed under CC BY-SA 4.0.

How to get an index?

- You have already benefited from indexes off-the-shelf

How to get an index?

- You have already benefited from indexes off-the-shelf
  - `PRIMARY KEY` constraint
  - `UNIQUE` constraint

How to get an index?

- You have already benefited from indexes off-the-shelf
  - `PRIMARY KEY` constraint
  - `UNIQUE` constraint
- Why does your DBMS create them automatically?

How to get an index?

Or, you can `CREATE INDEX`[1] on your own:

SQL Command Syntax

```
CREATE INDEX index_name
ON table_name (column_name [, ...]);
```

[1] https://www.postgresql.org/docs/current/sql-createindex.html

Index Taxonomy

- In terms of storage structure, is the index completely separated with the data records?
  - No $\implies$ **Integrated index**, e.g.
    - PK index in a MySQL InnoDB database
    - PK index in a SQL Server database
  - Yes $\implies$ **External index**, e.g.
    - Indexes in a PostgreSQL database
    - Indexes in a MySQL MyISAM database

[2] Someone may consider *primary index* as the alias of *integrated index*, and consider *secondary index* as the alias of *external index*. It depends on who you are talking with.

Index Taxonomy

- In terms of storage structure, is the index completely separated with the data records?
  - No $\Longrightarrow$ **Integrated index**, e.g.
    - PK index in a MySQL InnoDB database
    - PK index in a SQL Server database
  - Yes $\Longrightarrow$ **External index**, e.g.
    - Indexes in a PostgreSQL database
    - Indexes in a MySQL MyISAM database
- Does the index specify the order in which records are stored in the data file?
  - Yes $\Longrightarrow$ **Clustered index** (a.k.a. primary index[2])
  - No $\Longrightarrow$ **Non-clustered index** (a.k.a. secondary index)

---

[2] Someone may consider *primary index* as the alias of *integrated index*, and consider *secondary index* as the alias of *external index*. It depends on who you are talking with.

Index Taxonomy

- Does every search key in the data file correspond to an index entry?
  - Yes $\Longrightarrow$ **Dense Index**
  - No $\Longrightarrow$ **Sparse Index**

Index Taxonomy

- Does every search key in the data file correspond to an index entry?
  - Yes $\Longrightarrow$ **Dense Index**
  - No $\Longrightarrow$ **Sparse Index**
- Does the search key contain more than one attribute?
  - Yes $\Longrightarrow$ **Multi-key index** (Multi-column index)
  - No $\Longrightarrow$ **Single-key index** (Single-column index)

Index implementation

Data Structures for Indexes

- Ordered
  - *B*-tree and its variants (e.g. B^+-tree)
  - Generalizations of *B*-tree, e.g. *R*-tree
  - ...

Index implementation

Data Structures for Indexes

- Ordered
  - *B*-tree and its variants (e.g. B^+-tree)
  - Generalizations of *B*-tree, e.g. *R*-tree
  - ...
- Unordered
  - Hash tables
  - Bitmap
  - ...

Index implementation

Data Structures for Indexes

- Ordered
  - *B*-tree and its variants (e.g. B^+-tree)
  - Generalizations of *B*-tree, e.g. *R*-tree
  - ...
- Unordered
  - Hash tables
  - Bitmap
  - ...

Question

What is the common limitation of unordered indexes?

Index implementation: B-tree definition[3]

Definition

A *B-tree of order m* satisfies that

1. For every node, # of children $=$ # of keys $+ 1$

[3] Here we mainly follow D. E. Knuth's definition [2]. Note that the *B*-tree defined in *CLRS* [3] is slightly different. Read `https://stackoverflow.com/a/45826413` for further comparison.

Index implementation: B-tree definition[3]

Definition

A *B-tree of order m* satisfies that

1. For every node, # of children $=$ # of keys $+ 1$
2. **(Ordered)** For a node containing n keys $(K_1 < K_2 < K_3 < \cdots < K_n)$ with $n+1$ children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key k_{sub_i} in the sub-tree pointed by P_i satisfies that $K_i < k_{sub_i} < K_{i+1}$

[3]Here we mainly follow D. E. Knuth's definition [2]. Note that the B-tree defined in *CLRS* [3] is slightly different. Read `https://stackoverflow.com/a/45826413` for further comparison.

Index implementation: *B*-tree definition[3]

Definition

A *B-tree of order m* satisfies that

1. For every node, # of children $=$ # of keys $+ 1$
2. **(Ordered)** For a node containing n keys $(K_1 < K_2 < K_3 < \cdots < K_n)$ with $n+1$ children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key k_{sub_i} in the sub-tree pointed by P_i satisfies that $K_i < k_{sub_i} < K_{i+1}$
3. **(Multiway)** For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$, except that a root node may have less than $\lceil m/2 \rceil$ children

[3]Here we mainly follow D. E. Knuth's definition [2]. Note that the *B*-tree defined in *CLRS* [3] is slightly different. Read `https://stackoverflow.com/a/45826413` for further comparison.

Index implementation: B-tree definition[3]

Definition

A *B-tree of order m* satisfies that

1. For every node, # of children $=$ # of keys $+ 1$
2. **(Ordered)** For a node containing n keys $(K_1 < K_2 < K_3 < \cdots < K_n)$ with $n+1$ children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key k_{sub_i} in the sub-tree pointed by P_i satisfies that $K_i < k_{sub_i} < K_{i+1}$
3. **(Multiway)** For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$, except that a root node may have less than $\lceil m/2 \rceil$ children
4. **(Always balanced)** All leaves appear on the same level

---

[3] Here we mainly follow D. E. Knuth's definition [2]. Note that the B-tree defined in *CLRS* [3] is slightly different. Read `https://stackoverflow.com/a/45826413` for further comparison.

Index implementation: *B*-tree definition

Terminology

- $\lceil m/2 \rceil$ is the called the *minimum branching factor* (a.k.a. *minimum degree*) of the tree
- A B-tree of order m is usually called a *$\lceil m/2 \rceil$-m tree*, like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, ...

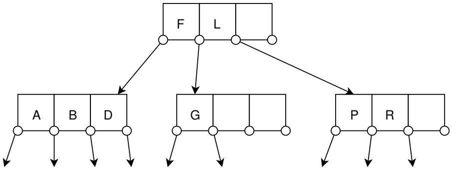


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

Index implementation: *B*-tree definition

Terminology

- $\lceil m/2 \rceil$ is the called the *minimum branching factor* (a.k.a. *minimum degree*) of the tree
- A B-tree of order m is usually called a $\lceil m/2 \rceil$*-m tree*, like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, ...

Remark

In practice, the order *m* is much larger (typically on the order of 100)

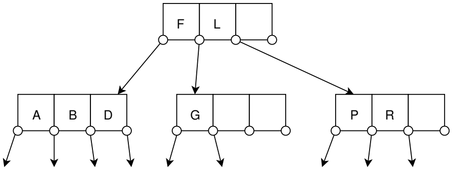


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

Index implementation: *B*-tree definition

Terminology

- $\lceil m/2 \rceil$ is the called the *minimum branching factor* (a.k.a. *minimum degree*) of the tree
- A B-tree of order m is usually called a *$\lceil m/2 \rceil$-m tree*, like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, ...

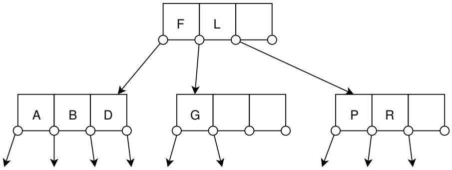


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

Remark

In practice, the order *m* is much larger (typically on the order of 100)

Question

Note that a *B*-tree usually waste some space. Why do we prefer *B*-tree to balanced BST (e.g. AVL tree, red-black tree, etc.)?

Index implementation: Why *B*-trees?

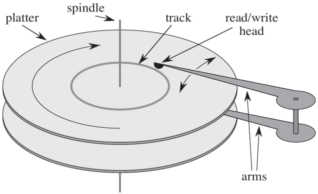


Figure: A typical disk drive (Retrieved from *CLRS* 3rd ed., pp. 485. [3])

Memory Hierarchy [4]

Technology	Access time
DRAM	50-70 ns
Magnetic disk	5-20 ms

Index implementation: Why *B*-trees?

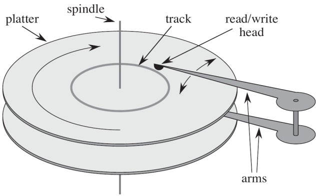


Figure: A typical disk drive (Retrieved from *CLRS* 3rd ed., pp. 485. [3])

Memory Hierarchy [4]

Technology	Access time
DRAM	50-70 ns
Magnetic disk	5-20 ms

- Locality $\longrightarrow$ Paging
  - disk page $\leftrightarrow$ *B*-tree node
  - *m* is chosen with consideration on the page size
  - # of disk I/O $\leftrightarrow$ height of *B*-tree

Index implementation: Why B-trees?

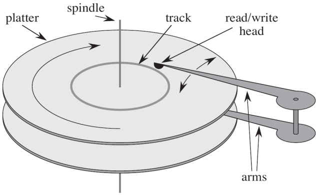


Figure: A typical disk drive (Retrieved from *CLRS* 3rd ed., pp. 485. [3])

Memory Hierarchy [4]

Technology	Access time
DRAM	50-70 ns
Magnetic disk	5-20 ms

- Locality $\longrightarrow$ Paging
  - disk page $\leftrightarrow$ B-tree node
  - m is chosen with consideration on the page size
  - # of disk I/O $\leftrightarrow$ height of B-tree
- Height of a B-tree

$$h \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right)$$

(What's the height of a B-tree of order 200 with two million search keys?)

Index implementation: *B*-tree operations

Tree operations:

- Search
- Insert
- Delete
- Update: Delete + Insert

Index implementation: *B*-tree operations

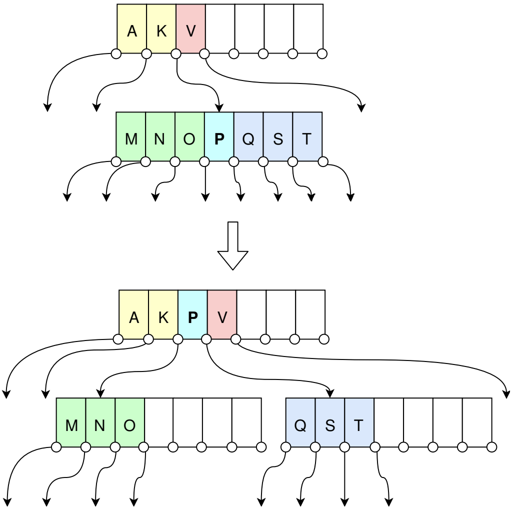
Tree operations:

- Search
- Insert
- Delete
- Update: Delete + Insert

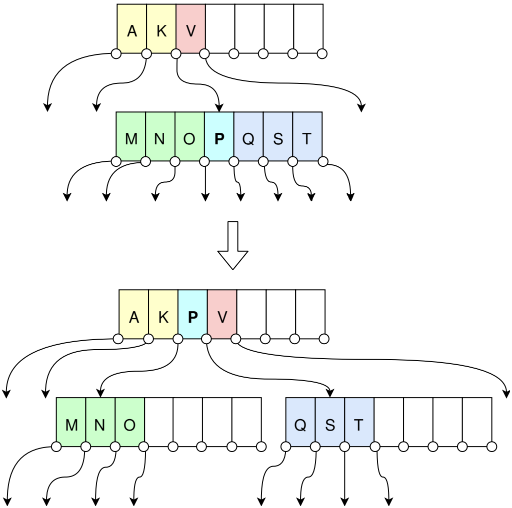
What is special in *B*-tree?

- Split
- Merge

Index implementation: Split a node in a *B*-tree



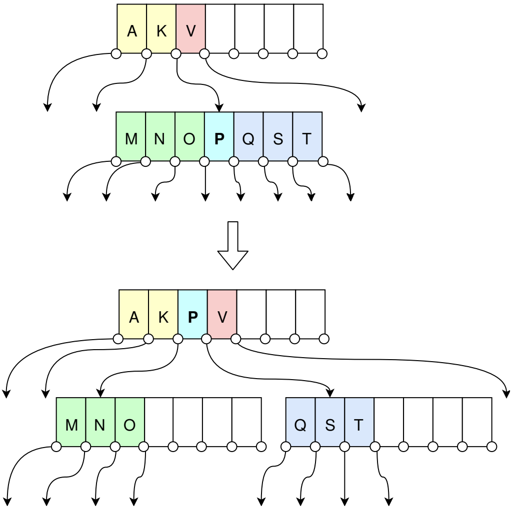
Index implementation: Split a node in a *B*-tree



Observation

In this example, the height of the *B*-tree doesn't change after splitting.

Index implementation: Split a node in a *B*-tree



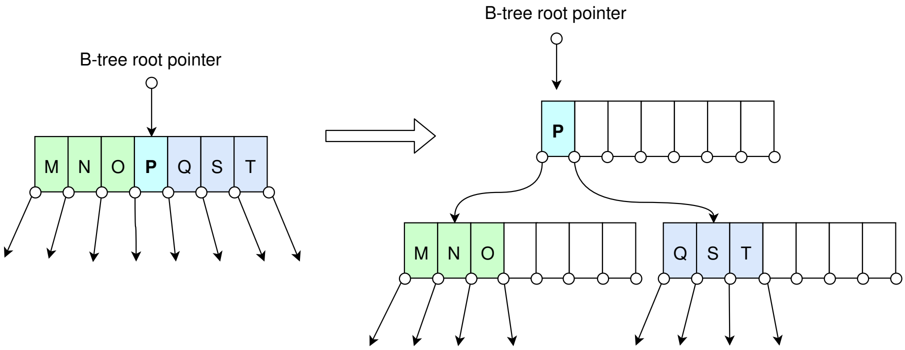
Observation

In this example, the height of the *B*-tree doesn't change after splitting.

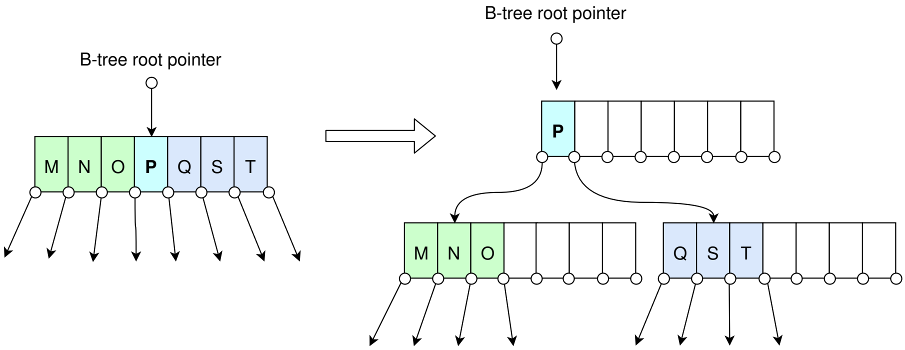
Question

What if the parent node is also full?

Index implementation: Split the root node of a *B*-tree



Index implementation: Split the root node of a *B*-tree



Remark

Note that the height of the *B*-tree is increased by one. This is the only way that a *B*-tree increases its height.

Index implementation: Delete a key in a *B*-tree

More complicated than insertion. May involve

- Rotation
- Merging

Suggestions:

- Watch the video [9].
- Read the related chapters [2, 3].

Index implementation: B^+-tree vs. B-tree

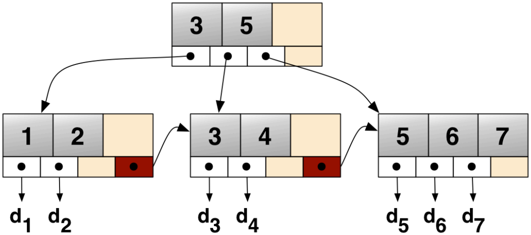


Figure: An example of B^+-tree (published by Grundprinzip under CC BY 3.0)

B-tree	B^+-tree
Data stored in every node	Data stored only in leaves
No sequential links	Leaves linked sequentially

Index implementation: Benefits of B^+-trees

Data stored only in leaves $\implies$ Larger m and smaller height

Index implementation: Benefits of B^+-trees

Data stored only in leaves	$\Longrightarrow$	Larger m and smaller height
Leaves linked sequentially	$\Longrightarrow$	Better support for range query

Negative impacts of indexes: Storage overhead

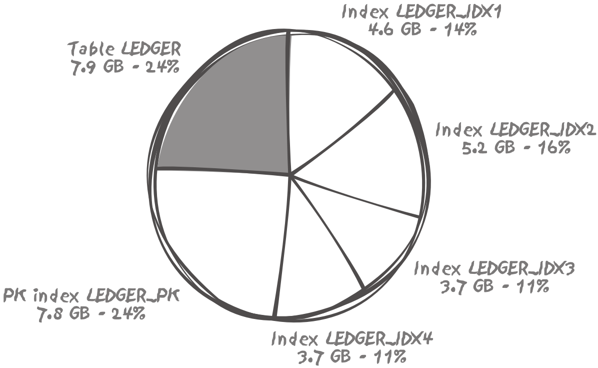


Figure: A real-life case: Data versus Index out of a 33 GB total (Retrieved from *The Art of SQL*, ch. 3, pp. 57. [5])

Reminder

The more indexes created, the more storage space consumed.

Negative impacts of indexes: Processing overhead

Reminder

Indexing may speed up data retrieval, but will bring overhead to `INSERT`, `UPDATE`, and `DELETE` operations.

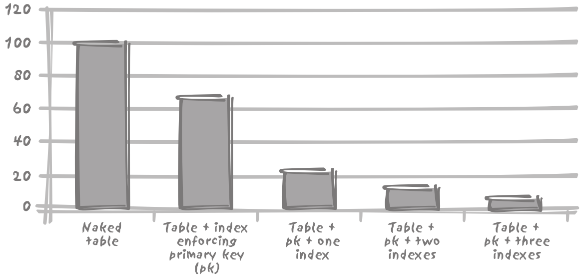


Figure: The impacts of indexes on insertion with Oracle (Retrieved from *The Art of SQL*, ch. 3, pp. 57. [5])

Negative impacts of indexes: Processing overhead

Reminder

Indexing may speed up data retrieval, but will bring overhead to `INSERT`, `UPDATE`, and `DELETE` operations.

Indexes and **Backup**:

- Logical backup
- Physical backup

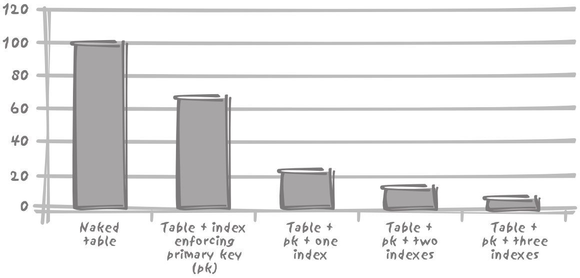


Figure: The impacts of indexes on insertion with Oracle (Retrieved from *The Art of SQL*, ch. 3, pp. 57. [5])

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<`, `<=`, `>`, `>=`, `BETWEEN`
  - `IN`
  - `EXISTS`
  - `LIKE` (prefix matching)

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<, <=, >, >=, BETWEEN`
  - `IN`
  - `EXISTS`
  - `LIKE` (prefix matching)
- Columns involved in foreign key constraints
  - Why indexing?

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<, <=, >, >=, BETWEEN`
  - `IN`
  - `EXISTS`
  - `LIKE` (prefix matching)
- Columns involved in foreign key constraints
  - Why indexing?
  - Think about two directions

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<, <=, >, >=, BETWEEN`
  - `IN`
  - `EXISTS`
  - `LIKE` (prefix matching)
- Columns involved in foreign key constraints
  - Why indexing?
  - Think about two directions
  - Will you update/delete those referenced records?

Index it or not: Where indexing may help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<, <=, >, >=, BETWEEN`
  - `IN`
  - `EXISTS`
  - `LIKE` (prefix matching)
- Columns involved in foreign key constraints
  - Why indexing?
  - Think about two directions
  - Will you frequently update/delete those referenced records?

Index it or not: Where indexing may not help

- Longest prefix matching of multi-key indexes
  - When specifying a composite primary key or creating a multi-key index, the order of columns matters.
  - If required, you may consider creating indexes for (col1, col2, col3), (col2), and (col3) respectively.

[4] The expression should be deterministic. For detailed usage, please refer to `https://www.postgresql.org/docs/11/indexes-expressional.html`

Index it or not: Where indexing may not help

- Longest prefix matching of multi-key indexes
  - When specifying a composite primary key or creating a multi-key index, the order of columns matters.
  - If required, you may consider creating indexes for (col1, col2, col3), (col2), and (col3) respectively.
- Function applied

```sql
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key;
```

[4] The expression should be deterministic. For detailed usage, please refer to https://www.postgresql.org/docs/11/indexes-expressional.html

Index it or not: Where indexing may not help

- Longest prefix matching of multi-key indexes
  - When specifying a composite primary key or creating a multi-key index, the order of columns matters.
  - If required, you may consider creating indexes for (col1, col2, col3), (col2), and (col3) respectively.
- Function applied

```
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key;
```

  - Add one more column to store the function value?

[4] The expression should be deterministic. For detailed usage, please refer to https://www.postgresql.org/docs/11/indexes-expressional.html

Index it or not: Where indexing may not help

- Longest prefix matching of multi-key indexes
  - When specifying a composite primary key or creating a multi-key index, the order of columns matters.
  - If required, you may consider creating indexes for (col1, col2, col3), (col2), and (col3) respectively.
- Function applied

```
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key;
```

  - Add one more column to store the function value? 2NF violated

[4] The expression should be deterministic. For detailed usage, please refer to https://www.postgresql.org/docs/11/indexes-expressional.html

Index it or not: Where indexing may not help

- Longest prefix matching of multi-key indexes
  - When specifying a composite primary key or creating a multi-key index, the order of columns matters.
  - If required, you may consider creating indexes for (col1, col2, col3), (col2), and (col3) respectively.
- Function applied

```
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key;
```

  - Add one more column to store the function value? 2NF violated
  - Or, utilize *indexes on expressions* [4]

```
CREATE INDEX idx_name ON table1(function(col1));
```

[4] The expression should be deterministic. For detailed usage, please refer to `https://www.postgresql.org/docs/11/indexes-expressional.html`

Index it or not: Where indexing may not help

- Low selectivity

```
SELECT
  1.0 * COUNT(DISTINCT country) / COUNT(country)
FROM movies;
```

Index it or not: Where indexing may not help

- Low selectivity

```
SELECT
  1.0 * COUNT(DISTINCT country) / COUNT(country)
FROM movies;
```

- Small tables
    - Why?

Index it or not: Where indexing may not help

- Low selectivity

```
SELECT
  1.0 * COUNT(DISTINCT country) / COUNT(country)
FROM movies;
```

- Small tables
  - Why?

Reminder

- Full scan $\neq$ Bad scheme
- Index retrieval $\neq$ Good scheme

Contents

How `JOIN` works

Some widely used `JOIN` algorithms:

- Nested-loop join
- Hash join
- Sort-merge join

How JOIN works: Nested loop join

```
for each row in t1 match C1(t1)
    for each row in t2 match P(t1, t2)
        if C2(t2)
           add t1|t2 to the result
```

Improvement: Block nested-loop join

How `JOIN` works: Hash join

1. Create a hash table for the smaller table `t1` in the memory
2. Scan the larger table `t2`. For each record `r`,
    1. compute the hash value of `r.join_attribute`
    2. map to corresponding rows in `t1` using the hash table

How `JOIN` works: Sort-merge join (a.k.a. merge join)

1. Sort tables `t1` and `t2` respectively according to the join attributes
2. Perform an interleaved scan of `t1` and `t2`. When encountering a matched value, join the related rows together.

How `JOIN` works: Sort-merge join (a.k.a. merge join)

1. Sort tables `t1` and `t2` respectively according to the join attributes
2. Perform an interleaved scan of `t1` and `t2`. When encountering a matched value, join the related rows together.

Observation

When there are indexes on the join attributes, step 1, the most expensive operation, can be skipped because `t1` and `t2` are already sorted in this scenario.

Query optimizer

- Equivalence rules in Relational Algebra
  Example:

$$\sigma_{\theta_1 \wedge \theta_2}(R_1 \bowtie_\theta R_2) = (\sigma_{\theta_1}(R_1)) \bowtie_\theta (\sigma_{\theta_2}(R_2))$$

Query optimizer

- Equivalence rules in Relational Algebra
  Example:

$$\sigma_{\theta_1 \wedge \theta_2}(R_1 \bowtie_\theta R_2) = (\sigma_{\theta_1}(R_1)) \bowtie_\theta (\sigma_{\theta_2}(R_2))$$

- Cost estimation

Query optimizer

- Equivalence rules in Relational Algebra
  Example:

$$\sigma_{\theta_1 \wedge \theta_2}(R_1 \bowtie_\theta R_2) = (\sigma_{\theta_1}(R_1)) \bowtie_\theta (\sigma_{\theta_2}(R_2))$$

- Cost estimation
- Limitation

EXPLAIN the execution plan

- So, has the DBMS used the index?
- Which algorithm did it use on earth?

```
EXPLAIN SELECT *
FROM movies m INNER JOIN countries c
    ON m.country = c.country_code;
```

```
Hash Join (cost=6.16..201.16 rows=9538 width=49)
  Hash Cond: (m.country = c.country_code)
  ->  Seq Scan on movies m (cost=0.00..169.38 rows=9538 width=31)
  ->  Hash  (cost=3.85..3.85 rows=185 width=18)
        ->  Seq Scan on countries c (cost=0.00..3.85 rows=185 width=18)
```

EXPLAIN the execution plan

Some tools like *pgAdmin* can visualize the execution plan for you

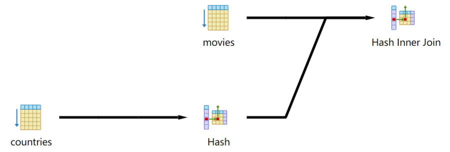


Figure: Execution plan visualized by pgAdmin 4

Reminder

- Full scan $\neq$ Bad scheme; Index retrieval $\neq$ Good scheme.
- When data stored are different, which implies different statistics, DBMS may have different execution plan for the same query.

Contents

Parameterized SQL

Parsing SQL costs time $\longrightarrow$ DBMS cache parsed SQL in memory

Parameterized SQL

Parsing SQL costs time $\longrightarrow$ DBMS cache parsed SQL in memory

Essentially the same statement, but caching doesn't help :(

```
String sql = "INSERT␣INTO␣students(name)␣VALUES␣("
        + stuName + ")";
```

Parameterized SQL

Parsing SQL costs time → DBMS cache parsed SQL in memory

Essentially the same statement, but caching doesn't help :(

```
String sql = "INSERT␣INTO␣students(name)␣VALUES␣("
        + stuName + ")";
```

Solution: variable binding

```
String sql = "INSERT␣INTO␣students(name)␣VALUES␣(?)";
PreparedStatement insert = con.prepareStatement(sql);
// ...
insert.setString(1, student.getName());
insert.executeUpdate();
con.commit();
```

Parameterized SQL

Why SQL parameterization is so important? Avoid SQL injection!

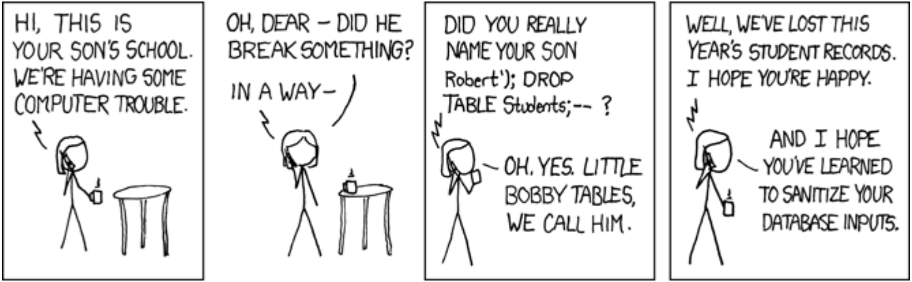


Figure: Exploits of a mom (Retrieved from `https://xkcd.com/327/`)

字符拼接一时爽，删库脱库火葬场!

Caching fetched data

Some DBMS can even cache query result. (Or you can consider other solutions like Memcached.)

Useful if

1. High-read
2. Low-write

Contents

Performance consideration beyond a single query

- Overhead outside the DBMS
  - TCP/IP protocol stack overhead
  - Context switch
- Multiple small operations vs. Batch operation
- Hardware consideration

Bibliography I

[1] A. Siberschatz, H. F. Korth, and S. Sudarshan, "Indexing and hashing," in *Database System Concepts*, 6th ed. New York: McGraw-Hill, 2011, ch. 11, pp. 475-536.

[2] D. E. Knuth, "Multiway trees," in *The Art of Computer Programming*, vol. 3, 2nd ed. Willard, Ohio: Addison-Wesley, 1998, ch. 6, sec. 6.2.4, pp. 481-489.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "B-tree," in *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009, ch. 18, pp. 277-289.

Bibliography II

[4] D. A. Patterson and J. L. Hennessy, “Memory technologies,” in *Computer Organization and Design: the hardware/software interface*, 5th ed. Waltham, MA: Elsevier Inc., 2011, ch. 5, sec. 5.2, pp. 378-383.

[5] S. Faroult and P. Robson, “Tactical Dispositions: Indexing,” in *The Art of SQL*, Sebastopol, CA: O’Reilly Media, Inc., 2006, ch. 3, pp. 55-74.

[6] 张金鹏, 张成远, and 季锡强, “数据结构和算法,” in *MariaDB 原理与实现*, Beijing, China: Posts & Telecom Press, 2015, ch. 9, pp. 197-218.

[7] “Indexes,” in *PostgreSQL 11 Documentation*. Available at `https://www.postgresql.org/docs/11/indexes.html`.

Bibliography III

[8] E. Demaine, "Lecture 5: Amortization: Amortized Analysis", *MIT OpenCourseWare, 6.046 Design and Analysis of Algorithms, Spring 2015*. Available at `https://youtu.be/3MpzavN3Mco`.

[9] A. S. Biswas, "Recitation 2: 2-3 Trees and B-trees", *MIT OpenCourseWare, 6.046 Design and Analysis of Algorithms, Spring 2015*. Available at `https://youtu.be/TOb1tuEZ2X4`.