ITS SEMANGAT BARU

ITS Institut Teknologi Sepuluh Nopember

Database Performance and Indexes

Database Performance

- Many situations where query performance needs to be improved
  - e.g., as data size grows, query performance degrades and tuning needs to be performed
  - Extreme cases: data warehouses with millions or billions of rows to aggregate and summarize
- To optimize queries effectively, we must understand what the database is doing under the hood
  - e.g., "Why are correlated subqueries slow to evaluate?"
    - Because an inner query must be evaluated for each row considered by the outer query. Thus, a good idea to avoid!

Disk Access

- First rule of database performance:

  Disk access is the most expensive thing databases do!
- Accessing data in memory can be 10-100ns
- Accessing data on disk can be up to 10s of ms
  - That's 5-6 orders of magnitude difference!
  - Even solid-state drives are 10s-100s of μs (1000x slower)
- Unfortunately, disk IO is usually unavoidable
  - Usually, the data simply doesn't fit into memory…
  - Plus, the data needs to be persistent for when the DB is shut down, or when the server crashes, etc.
- DBs work very hard to minimize the amount of disk IO

Planning and Optimization

- When the query planner/optimizer gets your query:
  - It explores many equivalent plans, estimating their cost (primarily IO cost), and chooses the least expensive one
  - Considers many options in evaluating your query:
    - What access paths does it have to the data you want?
    - What algorithms can it use for selects, joins, sorting, etc.?
    - What is the nature of the data itself?
      - i.e. statistics generated by the database, directly from your data
- The planner will do the best it can
  - Sometimes it can't find a fast way to run your query
  - Also depends on sophistication of the planner itself
    - e.g. if planner doesn't know how to optimize certain queries, or if executor doesn't implement very advanced algorithms

Table Data Storage

- Databases usually store each table in its own file
- File IO is performed in fixed-size blocks or pages
  - Common page size is 4KB or 8KB; can often tune this value
  - Disks can read/write entire pages faster than small amounts of bytes or individual records
  - Also makes it much easier for the database to manage pages of data in memory
    - The buffer manager takes care of this very complicated task
- Each block in the file contains some number of records
- Frequently, individual records can vary in size…
  - (due to variable-size types: VARCHAR, NUMERIC, etc.)

Table Data Storage (2)

- Individual blocks have internal structure, to manage:
  - Records that vary in size
  - Records that are deleted
  - Where and how to add a new record to the block, if there is space for it
- The table file itself also has internal structure:
  - Want to make sure common operations are fast!
  - "I want to insert a new row. Which block has space for it, or do I have to allocate a new block at the end of the file?"

Record Organization

- Should table records be organized in a specific way?
- Example: records are kept in sorted order, using a key
  - Called a sequential file organization
  - Would be much faster to find records based on the key
  - Would be much faster to do range queries as well
  - *Definitely* complicates the storage of records!
    - Can't predict order records will be added or deleted
    - Often requires periodic reorganization to ensure that records remain physically sorted on the disk
- Could also hash records based on some key
  - Called a hashing file organization
  - Again, speeds up access based on specific values
  - Similar organizational challenges arise over time…

Record Organization (2)

- The most common file organization is random!
  - Called a <u>heap file organization</u>
  - Every record can be placed anywhere in the table file, wherever there is space for the record
  - Virtually all databases provide heap file organization
  - Usually perfectly sufficient, except for most demanding applications

Heap Files and Queries

- Given that DBs normally use heap file organization, how does the DB evaluate a query like:

```
SELECT * FROM account
WHERE account_id = 'A-591';
```

- A simple approach:
  - Search through the entire table file, looking for all rows where value of *account_id* is A-591
  - This is called a file scan, for obvious reasons
- This will be slow, but it's all we can do so far…
- Need a way to optimize accesses like this

Table Indexes

- Most queries use a small number of rows from a table
  - Need a faster way to look up those values, besides scanning through entire data file
- Approach: build an index on the table
  - Each index is associated with a specific column or set of columns in the table, called the search key for the index
  - Queries involving those columns can often be made much faster by using the index on those columns
  - (Queries not using those columns will still use a file scan)
- Index is always structured in some way, for fast lookups
- Index is much smaller than the actual table itself
  - Much faster to search within the index (fewer IO operations)

Index Implementations

- Indexes are usually stored in files separate from the actual table data
  - Indexes are also read/written as blocks
- Indexes use <u>record pointers</u> to reference specific records in the table file
  - Simply consists of the block number the record is in, and the offset of the record within that block
- Index records contain values (or hashes), and one or more pointers to table records with those values

Index Characteristics

- Many different varieties of indexes, with different access characteristics
  - What kind of lookup is most efficient for the kind of index?
  - How costly is it to find a particular item, or a set of items?
    - e.g. a query retrieving records with a range of values
- Indexes do impose both a time and space overhead
  - Indexes must be kept up to date! Frequently, they slow down update operations, while making selects faster.
- Different kinds of indexes impose different overheads:
  - How much time to add a new item to the index? – insertion time
  - How much time to delete an item from the index? – deletion time
  - How much additional space does the index take up? – space overhead

Index Characteristics (2)

- Two major categories of indexes:
  - Ordered indexes keep values in a sorted order
    - E.g., author catalog in library.
  - Hash indexes divide values into bins, using a hash function
- Many variations within these two categories!

Dense Index Files

- Dense index — Index record appears for every search-key value in the file
  - It includes every single value from the source column(s).
- Faster lookups, but a larger space overhead.

Dense Index Files (2)

- E.g., index on *ID* attribute of *instructor* relation

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Dense Index Files (3)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - It only includes some of the values.
  - Lookups require searching more records, but index is smaller.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K*, we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

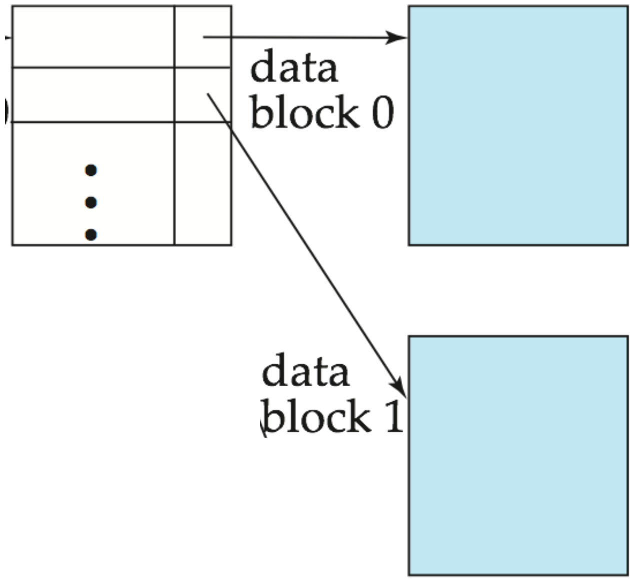
Sparse Index Files (2)

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sparse Index Files (2)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



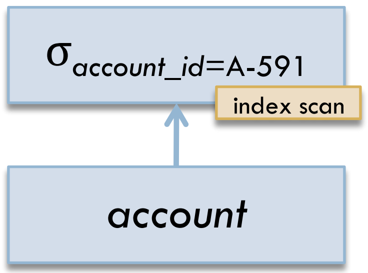
Indexes and Queries

- Indexes provide an alternate access path to specific records in a table
  - If looking for a specific value or range of values, use the index to find where to start looking in the table file
- Query planner looks for indexes on relevant columns when optimizing your query
- Query from before:

```
SELECT * FROM account
WHERE account_id='A-591';
```

- If there is an index on *account_id* column, planner can use an index scan instead of a file scan
  - Execution plan is annotated with these kinds of details

Execution Plan:



Keys and Indexes

- Databases create many indexes automatically
  - DB will create an index on the primary key columns, and sometimes on foreign key columns too
  - Makes it much faster for DB to enforce key and referential integrity constraints
- Many of your queries already use these indexes!
  - Lookups on primary keys, and joins on primary/foreign key columns
- Sometimes queries use columns that don't have indexes
  - e.g., SELECT * FROM account WHERE balance >= 3000;
- How do we tell what indexes the DB uses for a query?
- How do we create additional indexes on our tables?

EXPLAIN Yourself

- Most databases have an `EXPLAIN`-type command
  - Performs query planning and optimization phases, then outputs details about the execution plan
  - Reports, among other things, what indexes are used
- MySQL `EXPLAIN` command:

```
EXPLAIN SELECT * FROM account
WHERE account_id = 'A-591';
```

```
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table   | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | account | const | PRIMARY       | PRIMARY | 17      | const |    1 |       |
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------+
```

MySQL EXPLAIN (2)

- More interesting result with a different account ID:

```
EXPLAIN SELECT * FROM account
WHERE account_id = 'A-000';
```

```
+----+-------------+-------+-----+-----------------------------------------+
| id | select_type | table | ... | Extra                                   |
+----+-------------+-------+-----+-----------------------------------------+
|  1 | SIMPLE      | NULL  | ... | Impossible WHERE noticed after reading const tables |
+----+-------------+-------+-----+-----------------------------------------+
```

  - MySQL planner uses the primary key index to discern that the specified ID doesn't appear in the *account* table!

MySQL EXPLAIN (2)

- Another query against *account*:

```
EXPLAIN SELECT * FROM account
WHERE balance >= 3000;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	account	ALL	NULL	NULL	NULL	NULL	60	Using where

  - No index available to use for this column

Adding Indexes to Tables

- If many queries reference columns that don't have indexes, and performance becomes an issue:
  - Create additional indexes on a table to help the DB
- Usually specified with `CREATE INDEX` commands
- To speed up queries on account balances:

```
CREATE INDEX idx_balance ON account (balance);
```

  - Database will create the index file and populate it from the current contents of the account relation
  - (this could take some time for really large tables…)
- Can also create multi-column indexes
- Can specify many options, such as the index type
  - Virtually all databases create BTREE indexes by default

Adding Indexes to Tables (2)

- MySQL allows you to specify indexes in the CREATE TABLE command itself
  - not many other DBs support this, so it's not portable.
- Any drawbacks to putting an index on account balances?
  - It's a bank. Account balances change all the time.
  - Will definitely incur a performance penalty on updates (but, it probably won't be terribly substantial…)

Verifying Index Usage

- Very important to verify that your new index is actually being used!
  - If your query doesn't use the index, best to get rid of it!

```
EXPLAIN SELECT * FROM account
WHERE balance >= 3000;
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | account | ALL  | idx_balance   | NULL | NULL    | NULL |   60 | Using where |
+----+-------------+---------+------+---------------+------+---------+------+------+-------------+
```

- Hmm, MySQL doesn't use the index for this query.
  - If other expensive queries use it, makes sense to keep it (e.g., the rank query would use this index)
  - Otherwise, just get rid of it and keep your updates fast

Indexes and Performance Tuning

- Adding indexes to a schema is a common task in many database projects
- As a performance-tuning task, usually occurs after DB contains some data, and queries are slow
  - Always avoid premature optimization!
  - Always find out what the DB is doing first!
- Indexes impose an overhead in both space and time
  - Speeds up selects, but slows down all modifications
- Always need to verify that a new index is actually being used by the database. *If not, get rid of it!*

Thank you!