

DATABASE PERFORMANCE AND INDEXES

FALL2018: CS121: Relational Databases

Database Performance

2

- 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!

Database Performance (2)

3

- Next two lectures will explore how most databases evaluate queries
 - ▣ Specifically, how are relational algebra operations implemented, and what optimizations do they employ?
 - ▣ As usual, there are always exceptions! (e.g. MySQL)
 - ▣ Important to be aware of, so you understand each DBMS' limitations
- Today, will concentrate more on data storage and access methodologies
- Next time, explore relational algebra implementations
 - ▣ These are built on top of topics covered today

Disk Access!

4

- 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 μ s (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

5

- 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

6

- 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)

7

- 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

8

- 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)

9

- The most common file organization is random! 😊
 - ▣ Called a heap file organization
 - ▣ 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

10

- 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

11

- 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 Characteristics

12

- 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?
 - ▣ How much time to delete an item from the index?
 - ▣ How much additional space does the index take up?

Index Characteristics (2)

13

- Two major categories of indexes:
 - ▣ Ordered indexes keep values in a sorted order
 - ▣ Hash indexes divide values into bins, using a hash function
- Many variations within these two categories!
- Example: dense vs. sparse indexes
 - ▣ A dense index includes every single value from the source column(s). Faster lookups, but a larger space overhead.
 - ▣ A sparse index only includes some of the values. Lookups require searching more records, but index is smaller.
- The indexes we are covering today are dense indexes
 - ▣ Heap files are in random order, so an index won't help us very much unless it includes every value from the table

Index Implementations

14

- Indexes are usually stored in files separate from the actual table data
 - ▣ Indexes are also read/written as blocks
 - (Same reasons as before...)
- Indexes use record pointers 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 Implementations (2)

15

- Virtually all databases provide ordered indexes, using some kind of balanced tree structure
 - ▣ B⁺-tree and B-tree indexes, typically referred to as “btree” indexes
- Some databases also provide hash indexes
 - ▣ More complex to manage than ordered indexes, so not very common in open-source databases
- Several other kinds of indexes as well:
 - ▣ Bitmap indexes – to speed up queries on multiple keys
 - Also less common in open-source databases
 - ▣ R-tree indexes – to make spatial queries very fast
 - With ubiquity of geospatial data, quite common these days

B⁺-Tree Indexes

16

- A very widely used ordered index storage format
- Manages a balanced tree structure
 - ▣ Every path from root to leaf is the same length
 - ▣ Generally remains efficient for selects, even with inserts and deletes occurring
- Can consume significant space, since individual nodes can be up to half empty!
- Index updates for insert and delete can be slow...
 - ▣ Tree structure must be updated properly
- Performance benefits on queries more than outweigh these costs!

B⁺-Tree Indexes (2)

17

- Each tree node has up to n children
 - ▣ Simplification: n is fixed for the entire tree
- Each node stores n pointers and $n - 1$ values

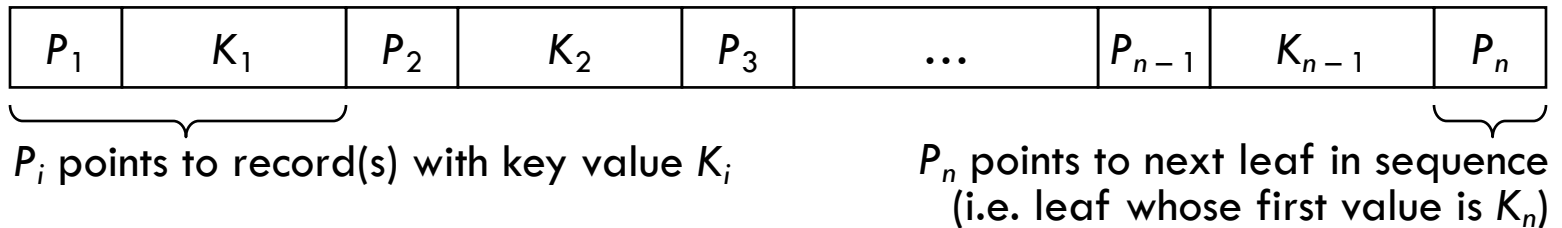
P_1	K_1	P_2	K_2	P_3	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-------	-------	-----	-----------	-----------	-------

- ▣ K_i are search-key values, P_i are record pointers
 - ▣ Values are kept in sorted order: if $i < j$ then $K_i < K_j$
 - ▣ All nodes (except root) must be at least half full
- Size of n depends on block size, search-key size, and record pointer size, but it is usually large!
 - ▣ Example: 4KB blocks, 4B record pointers, 4B integer keys
 - ▣ n will be >500 ! B⁺-tree indexes are shallow, broad trees.

B⁺-Tree Leaf Nodes

18

□ For leaf nodes:

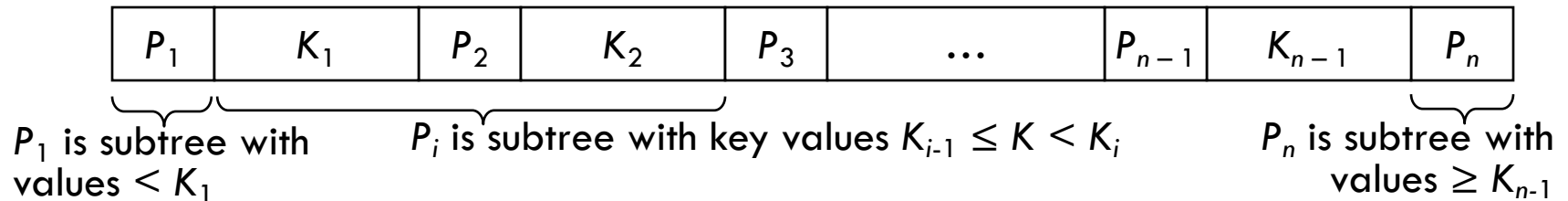


- Pointer P_i refers to record(s) with search-key value K_i
- If search key is a candidate key, P_i points to the record with key value K_i
- If search key isn't a candidate key, P_i points to a collection of pointers to all records with key value K_i
- No two leaves have overlapping ranges
 - Leaves can be arranged in sequential order
 - Pointer P_n points to the next leaf in sequential order

B⁺-Tree Non-Leaf Nodes

19

□ For non-leaf nodes:



▣ All pointers P_i refer to other B⁺-tree nodes

□ For $1 < i < n$:

▣ Pointer P_i points to subtree containing search-key values of at least K_{i-1} , but less than K_i

□ For $i = 1$ or $i = n$:

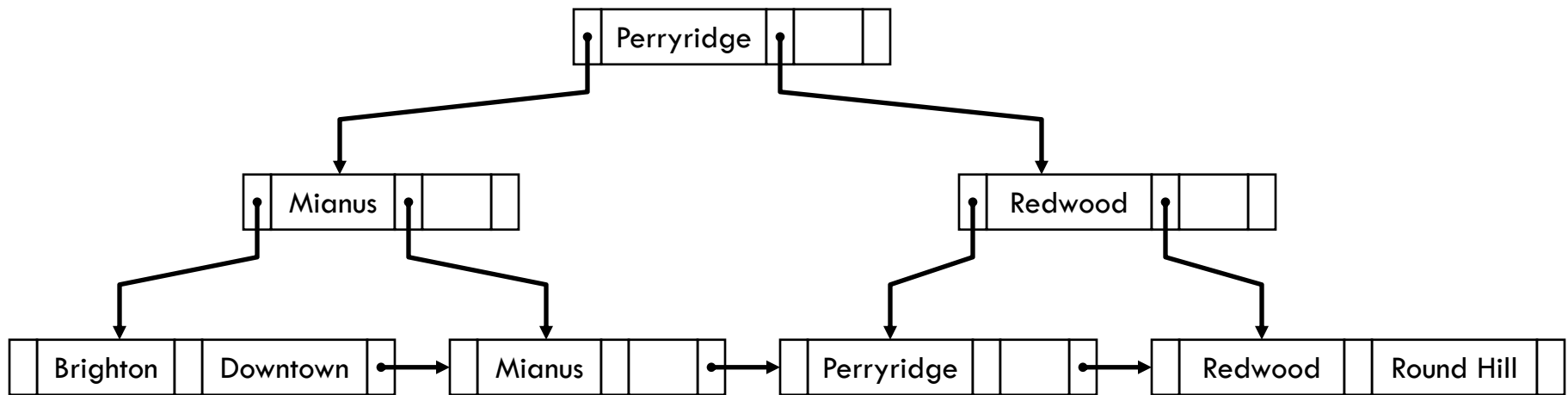
▣ Pointer P_1 points to subtree containing search-key values less than K_1

▣ Pointer P_n points to subtree containing search-key values at least K_{n-1}

Example B⁺-Tree

20

- A simple B⁺-tree, with $n = 3$



- Queries are straightforward
- Inserts may require one or more nodes to be split
- Deletes may require one or more nodes to be merged

B⁺-Trees and String Keys

21

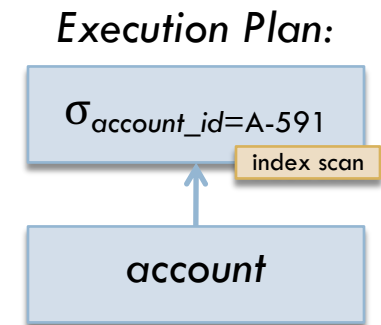
- String columns are problematic for indexing
 - ▣ Frequently specified to have large/variable-size values
 - ▣ Large keys reduce branching factor of each node, increasing tree depth and access cost
 - ▣ Large keys can also interfere with tree restructuring
- Simple solution: don't use the entire string! 😊
 - ▣ Can use prefix compression technique
 - ▣ Non-leaf nodes only store a prefix of the search string
 - ▣ Size of prefix must be large enough to distinguish reasonably well between values in each subtree
 - Otherwise, can't effectively narrow down records to consider

Indexes and Queries

23

- 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



Keys and Indexes

24

- 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

25

- 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	

- ▣ This query uses primary key index to look up the record
- ▣ MySQL knows that the result will be one row, or no rows

MySQL EXPLAIN (2)

26

- 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!
- 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

27

- 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)

28

- 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

29

- 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 on Large Values

30

- Large keys seriously degrade index performance
- Example: B-trees and B⁺-trees
 - ▣ Biggest benefit is very large branching factor of each node
 - ▣ Large key-values will dramatically reduce the branching factor, deepening the tree and increasing IO costs
- Can specify indexes on only the first *N* characters/bytes of a string/LOB value

```
CREATE INDEX idx_name ON customer (cust_name(5));
```

 - ▣ Only uses first five characters for customer-name index
 - ▣ If most values differ in first *N* bytes, index will be much smaller and faster for both updates and queries
 - ▣ If values don't differ much, index won't do much good

Indexes and Performance Tuning

31

- 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!*

Administrivia

32

- Next time: SQL Query Evaluation II
 - ▣ Overview of how most relational algebra operators are implemented, including common-case optimizations

- Midterm time is a-comin'...
 - ▣ Next Monday, November 5, is midterm review
 - ▣ Come to class, watch the video, get the slides, whatever.
 - ▣ Midterm will be available towards end of next week
 - ▣ No assignment due the week of the midterm