Database Performance Tuning for Application Developers (i.e. Indexing)

Bruce Cota
Unicon Inc

Why you should care

- It's Easy
- Impact can be dramatic
- Your DBA may not be doing it
- Should be “basic stuff” but often falls through cracks.

Why isn't your DBA doing it?

DBA certification process trains people for “generic” duties that don't require application knowledge. (backups, networking, etc)

Certification exams emphasize performance tuning through configuration parameters.

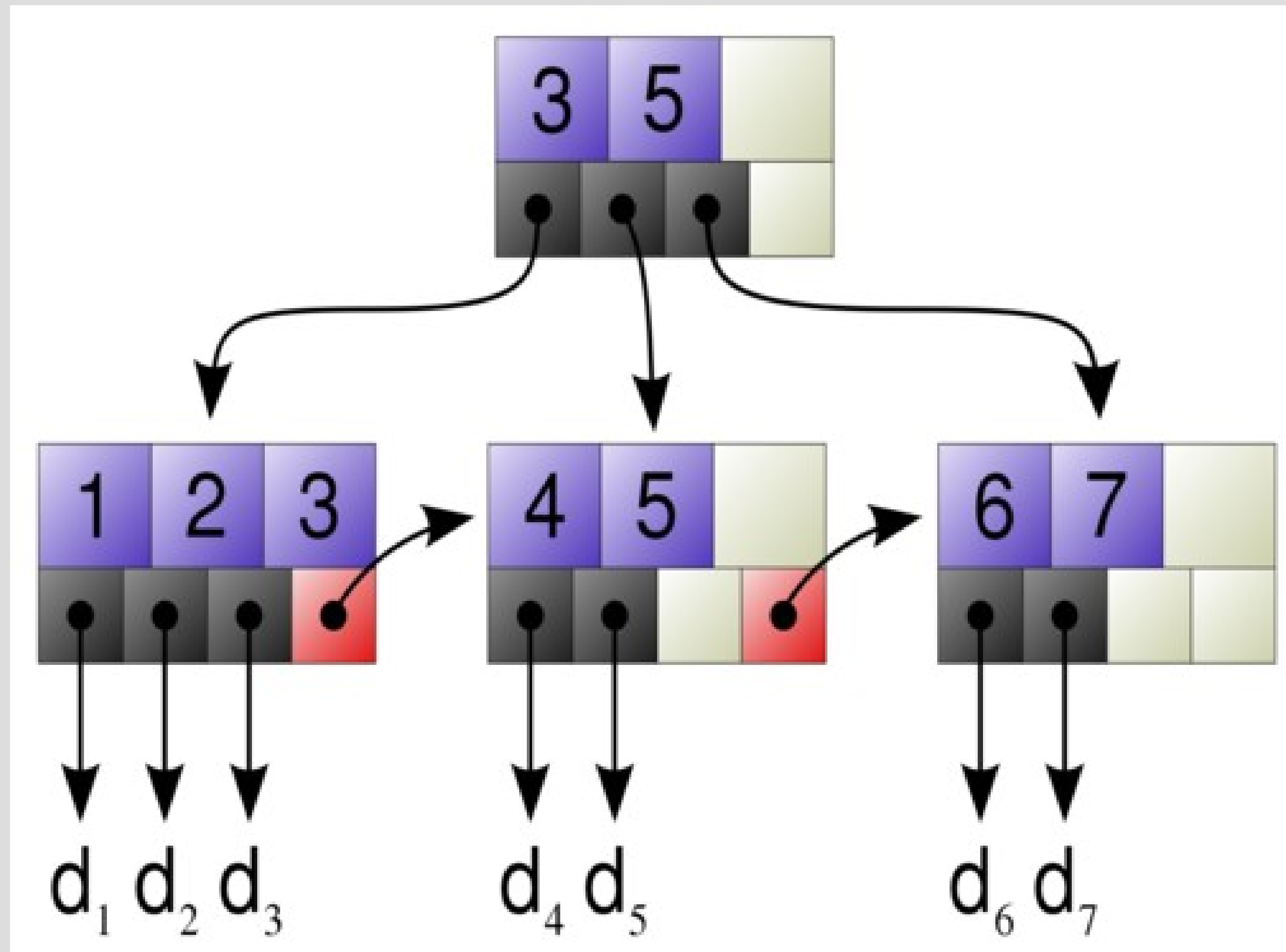
You need application specific knowledge.

Computer Science background is helpful.

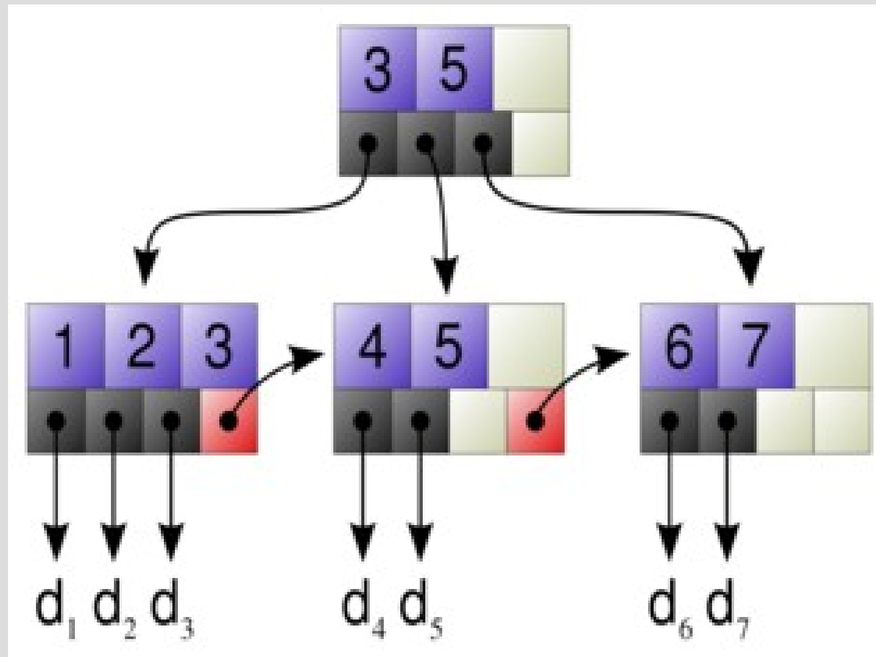
Goals

- Answer questions like
  - Can an index help with text searches?
  - How do I index a key like “(user_id, class_id)”
  - What indexes should I start with?
  - Is it bad to use strings as keys?
- Avoid Faux Pas like
  - It can't be slower, I didn't change any code!
  - Of course it's slower, the tables are bigger!

B+-Trees

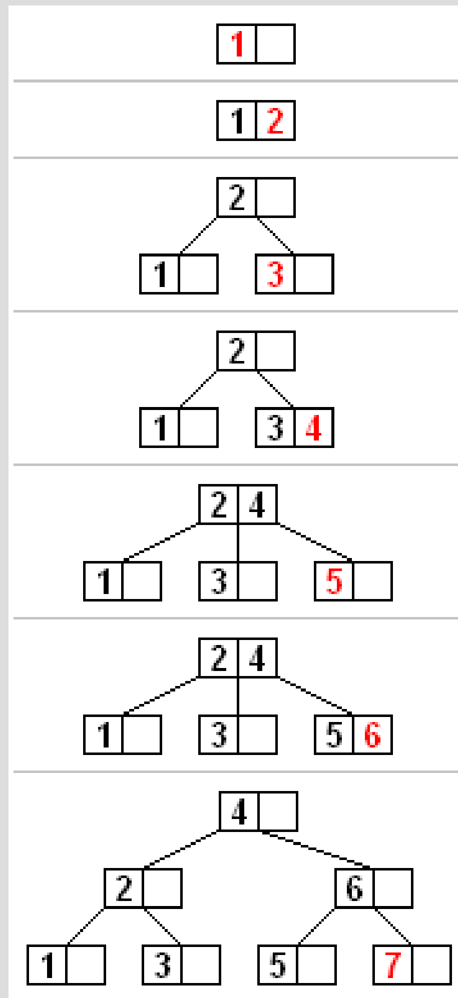


B+-Trees



Keys are linearly ordered.
Tree is balanced.
Nodes are “blocks”.
Blocks are not full.
Data rows outside the tree. (the “+” part)

Keeping B-Trees Balanced



- Key inserted into correct block.
- Full block splits, median key moves up.
- Tree grows when root block splits.
- Variants Exist

Searching in a B+-Tree

Search descends from the root node to a leaf. (Then jumps to data row.)

# of blocks processed == height of tree +1.

Height of **M**-ary tree with **N** blocks is proportional to $\log_M(N)+1$

Insert and Delete are also $O(\log_M(N))$

About the "M"

Search is $\mathbf{Log_M(N)}$ where **N** is the number of keys and **M** is number of keys per block.

Math Review: $\mathbf{Log_M(N) = Log(N) / Log(M)}$

- Search time decreases with **M** (keys per block)
- **M** may be large (4K-16K blocks, 4-8 byte keys)
- Worth avoiding huge keys (e.g. Varchar2(2000))
- But penalty is logarithmic, so don't worry too much.

Real World Examples (Actual Production Tables)

- Table1 (5 byte key)
  - 1,177 rows
  - 64 data blocks
  - Index tree height: 1 – searches process 2 blocks
- Table2
  - 22,651 rows
  - 115 data blocks
  - Index height: 1

Examples

- Table3
  - 208,132 rows
  - 731 data blocks
  - Index height: 1
- Table4 (10 byte key)
  - 542,257 rows
  - 74,750 data blocks (fat rows!)
  - Index height: 2

Examples

- Table5
  - 49,788,194 rows
  - 510,262 data blocks
  - Index height 2
- Table6 (10 byte key)
  - 1,148,048,215 rows
  - 2,894,272 data blocks
  - Index height 3

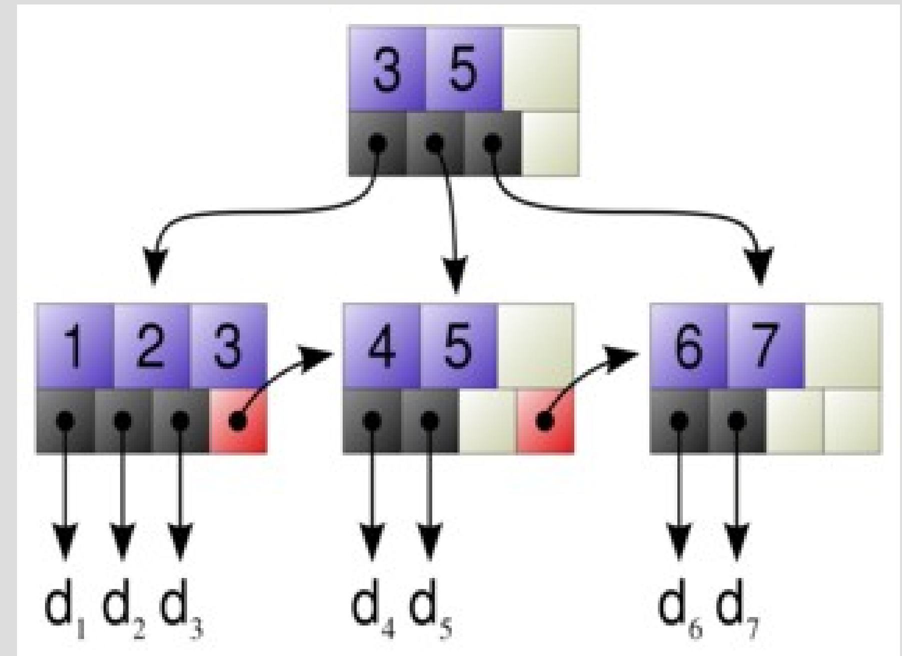
  Note: table has 1 billion rows and nearly 3 million blocks, but indexed search only examines 4 blocks. (Table scan would examine every block.)

Range Searches
(.. where x > 3 and x < 6)

First search for lower bound.

Scan leaf nodes until upper bound.

Time is **$Log_M(N)+O(K)$** where **k** is number of rows in range.



Why B-Tree is Default

- Databases are block oriented
  - Blocks processed during a query is sometimes the best performance metric.
- B-Tree supports
  - Concurrent transactions
    - (but fear the “hot page” syndrome)
  - Range searches
- Alternatives include Hash Tables, Bitmaps

Composite Indexes

- Indexes on composite keys like "`(User_id,Class_id)`"
- B-Tree requires linear ordering on key
- "Dictionary Order" -- `User_id` determines ordering, `Class_id` is "tie breaker"
- e.g. (10,1000) < (20,500) < (20, 750)
- Can be extended to any number of columns.

Prefix Searches

How can a composite index on `(User_id,Class_id)` be used for a query like "`SELECT * FROM User_Class WHERE User_id=X`"

Prefix Searches

Answer: A prefix search is essentially a range search equivalent to “`SELECT * FROM User_Class WHERE User_id=X and class_id > MIN and class_id < MAX`”

A value for the first column(s) allow the first result row to be found in log time.

Then scan leaf blocks until prefix changes.

Non-Prefix Searches

- How can an index on `(User_id,Class_id)` be used for a query like "`SELECT * FROM User_Class WHERE Class_id = X`"
- Since Class_id is a "tie breaker" in the linear ordering, rows with the same Class_id will be scattered around the index. (Range search won't work)
- A technique exists, not supported by all vendors.

Non-Prefix Searches
"Fast Full Index Scans"

Answer: Search every leaf block.

If keys are smaller than complete rows, there will be fewer leaf blocks than data blocks.

Exam_Result has 510,262 data blocks, and a composite index with 172,544 leaf blocks.

Faster but still `O(N)`

If N is growing, should an `O(N)` algorithm *ever* be used for a customer facing web page?

Text Searches with B-Trees

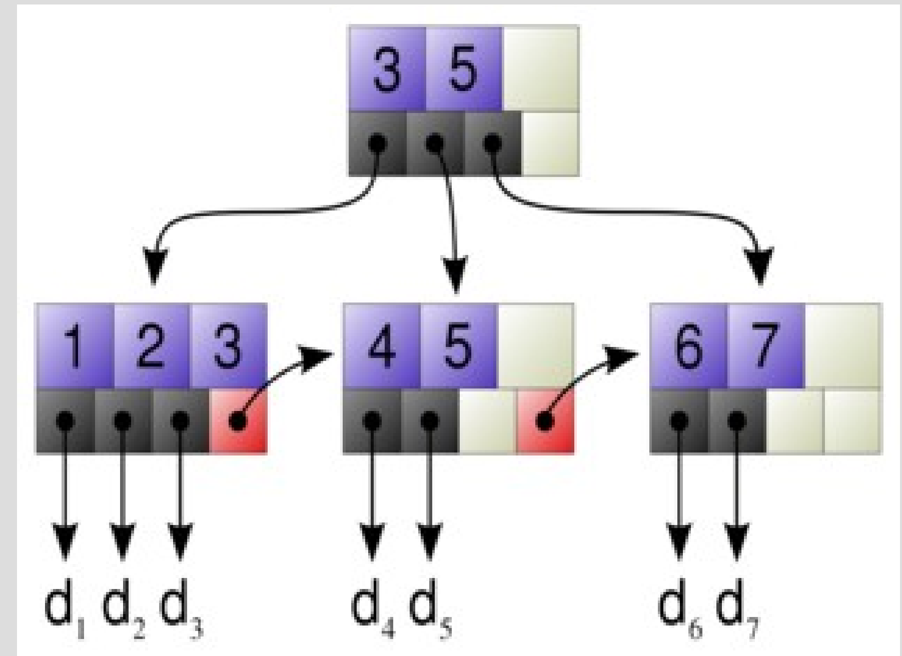
- "`SELECT * FROM t WHERE fname LIKE 'blah%'`" (pattern matching in sql) is a prefix search.
  - `O(Log(N))` time if `fname` is indexed
- "`SELECT * FROM t WHERE name LIKE '%blah'`" is *not* a prefix search. (`O(N)` time)
- For non-prefix text searches, consider something else (e.g. Lucene)

Functional Indexes

- Watch out for case sensitivity . Indexing `user.email` won't help with "`SELECT * FROM user WHERE LOWER(email)=LOWER(?)`"
- Most(?) vendors support *functional indexes*
  - CREATE INDEX email_idx ON user(LOWER(email))
- Ditto for mathematical functions, e.g. "`SELECT * FROM position WHERE sin(x)=?`"
- Values in the B-Tree must be the values you search on.

When Table Scans are Faster than Indexed Searches.

- Extra pointer to data row.
- Each row returned == +1 block processed.
- Same block may be processed over and over.
- Sequential scans are optimized.
- Table scan is faster when a high percentage of rows are returned. (4%?)
- Avoid indexes with a small number of key values.



What number of keys is "small"?

I've heard specific numbers, like "25".

I think a fixed number of key values is "small" for a growing table.

Example: Consider indexing Exam Results by grade.

Bitmap indexes work for this, but they have other limitations.

Why Your Database Seems to Hate You

The *query optimizer* **heuristically** chooses a *query plan*, (what indexes to use, etc).

Best case indexed search is often ***much*** better than best cased table scan.

Worst case table scan is often better than worst case indexed search.

For complex queries, the optimizer may choose a table scan when a web app requires indexed search.

Avoiding Pain

- Optimizer requires data distribution statistics to make decisions These are updated by a DBA operation.
- Should be updated when data hits some kind of “steady state”.
- Vendors advise frequently updating statistics, but this is risky because query plans change
- If it's not broken, why fix it??

Don't Index Everything

Indexes take a lot of space and memory.

When an indexed column is updated, the index must be updated.

Frequently updating multiple indexes on a “hot table” causes contention, may cause queries to fail due to deadlock.

Choosing Indexes

- Method 1: Rely on testing with DBA monitoring and advising.
- Method 2: Design indexes from a data model and application knowledge.
  - More stable across releases and datasets.
  - Test with DBA monitoring as validation.

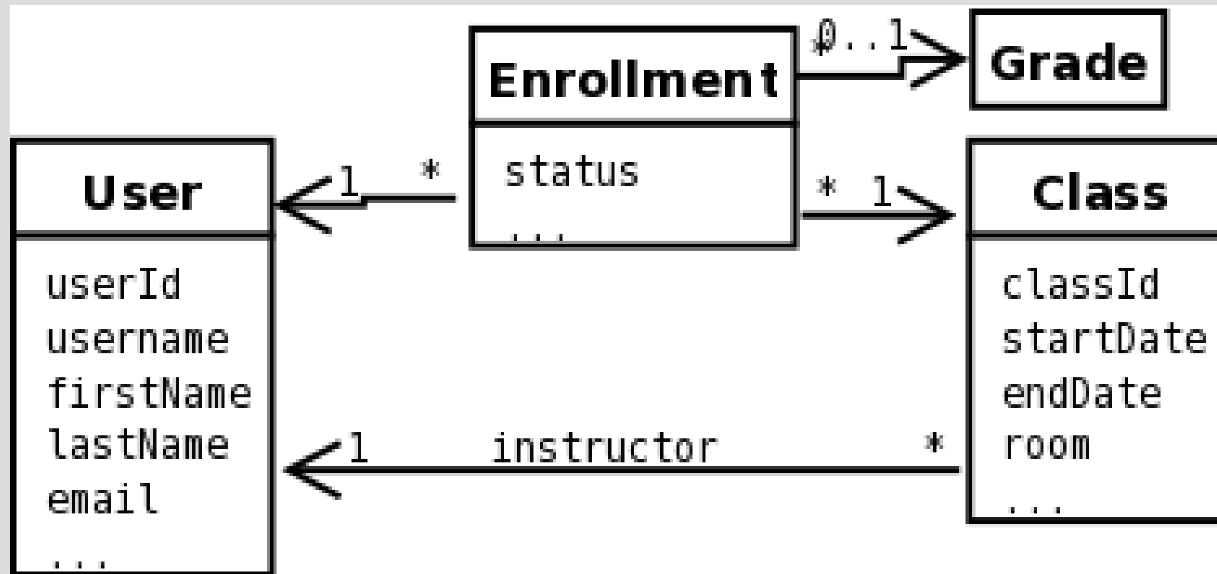
Choosing Indexes

Index all primary keys and foreign keys. (Requires that you have identified these.)

Identify searchable columns (typically compared to text fields filled in by users.)

Consider eliminating foreign keys from “big” to “small” tables. (Not enough key values.)

A Simple Example



Primary keys are User.userId, Class.classId, and TBD.
Foreign Keys are Enrollment->User, Enrollment->Grade, Enrollment->Class, and Class->User (instructor)
Foreign keys are the “traversal paths” through the model. (They are navigable in both directions.)

Indexes on User

```
User
+getUserId(): int
+getUserName(): String
+getFirstName(): String
+getLastName(): String
+getEmail(): String
+getEnrollments(): Set<Enrollment>
+getClasses(): List<Class>
```

```
User
User_Id: number(9) primary key
username: varchar(100) not null unique
First_Name: varchar(100)
Last_Name: varchar(100)
email: varchar(100)
```

- `User_Id` is declared primary key, auto-indexed.
- `Username` is probably filled in on a login form, so should be indexed.
- Declaring "`unique`" creates index.
- Index `email`, etc if they are searchable. Remember text-searching limitations.

Indexes on Class

```
Class
+getClassId()(): int
+getInstructor(): User
+getStartDate(): Date
+getEndDate(): Date
+getRoom(): String
+getEnrollments(): Set<Enrollment>
```

```
Class
Class_Id: number(9) primary key
instructor_id: number(9) references User
start_date: date
end_date: date
room: varchar(100)
```

`Class_id` is primary key

`instructor_id` is a foreign key (to User) and should be indexed.

A foreign key in Java (EJB) is an attribute whose value is another entity.

Attributes whose values are collections of children don't show up in database tables.

Indexes on Grade

```
Grade
+getGradeId(): int
+getLetter(): char
+isPassing(): boolean
+getMinPercent(): int
```

```
Grade
Grade_Id: number(3) primary key
letter: char(1) not null
is_passing: boolean not null
min_percent: number(2) not null
```

A *value table*.

Primary key is necessary, but probably too small for other indexes.

Enrollment 1.0

```
Enrollment
+getEnrollmentId(): int
+getUser(): User
+getClass(): Class
+getGrade(): Grade
+getStatus(): String
```

```
Enrollment
Enrollment_Id: number(9) primary key
User_Id: number(9) not null references User
Class_id: number(9) not null references Class
Grade_Id: number(9) not null references Grade
status: varchar(100) not null
```

- Added `Enrollment_ID` as primary key
- `User_Id` and `Class_Id` should be indexed as foreign keys
- Probably aren't enough distinct `Grade_Id`'s to be worth indexing.

Enrollment V 2.0

Enrollment

+getUser(): User
+getClass(): Class
+getGrade(): Grade
+getStatus(): String

Enrollment

User_Id: number(9) not null references User
Class_id: number(9) not null references Clas
Grade_Id: number(9) not null references Grac
status: varchar(100) not null

- Enrollment is an *association entity* and therefore *dependent*.
- `(user_id, class_id)` `(or vice versa)` is a good (composite) primary key.
- Must index `class_id` separately, but no need to index `user_id`.

Why Index the Foreign Keys?

- Foreign keys are “traversal paths”
- For example, counting number of students who have taken a class from Dr. Fred
  - Find user_id “x” of Dr. Fred
  - For each class “cl” with instructor_id = “x”
    - Count enrollments “e” where e.class_id = cl.class_id
- That is called a “nested loop join”

Conclusions

- Application developers should take some responsibility for database performance.
- Indexes are (usually) good.
- Table scans are (usually) bad.
- You can identify the indexes you need by understanding your application and understanding data structures.