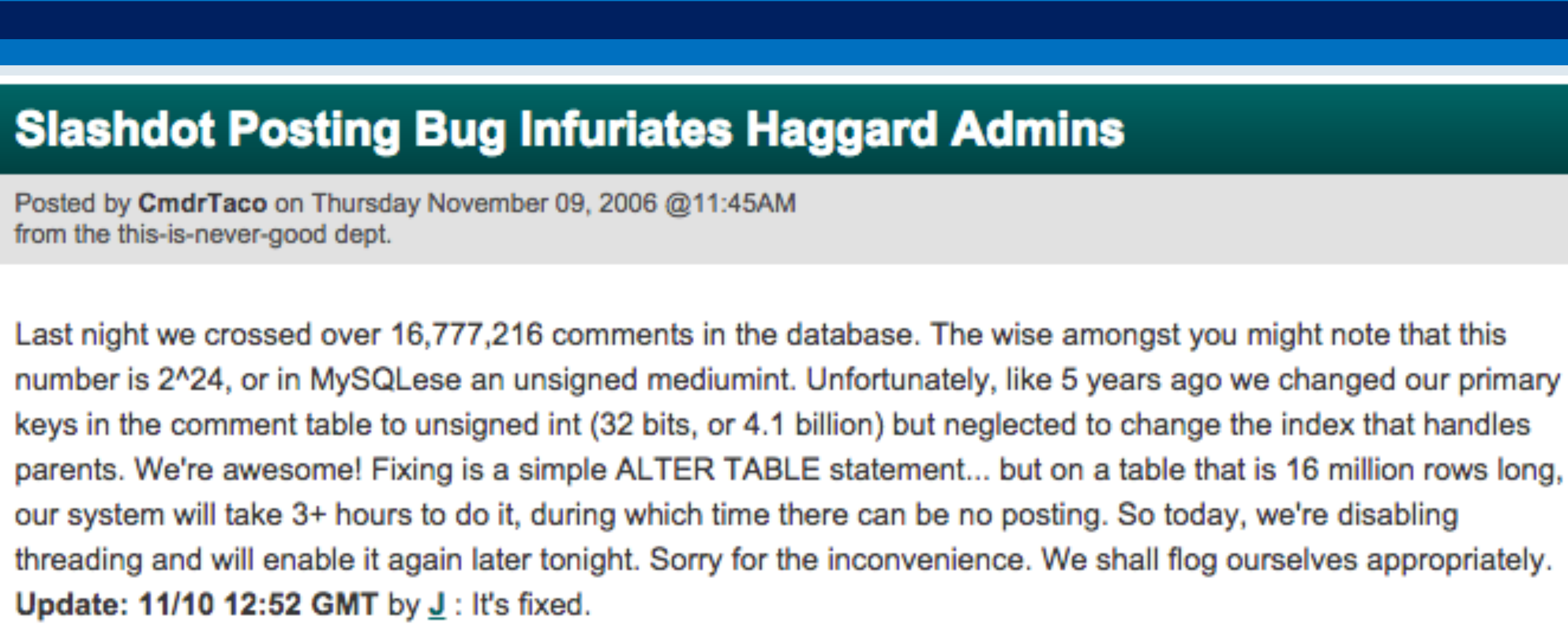
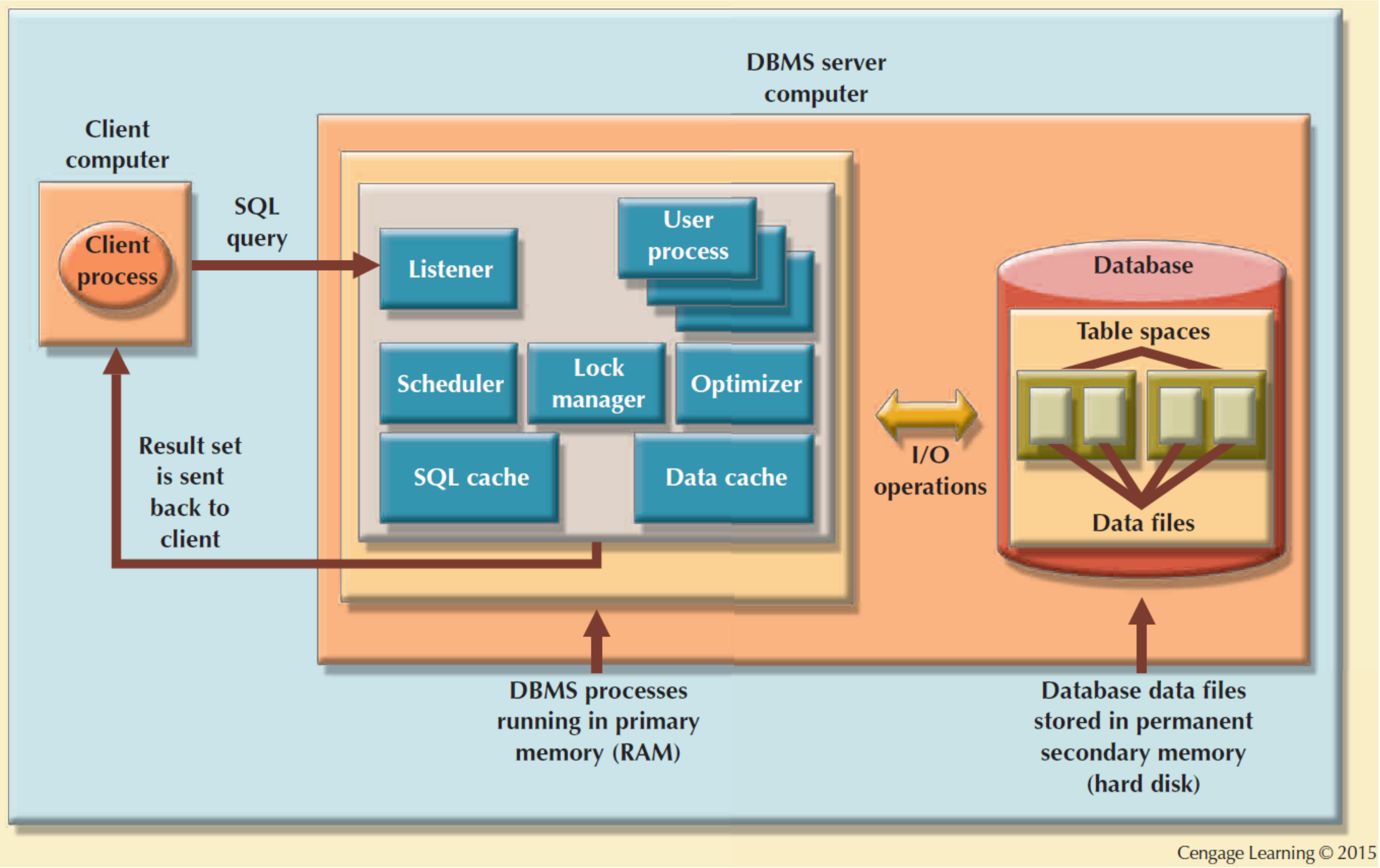


Slashdot Posting Bug Infuriates Haggard Admins

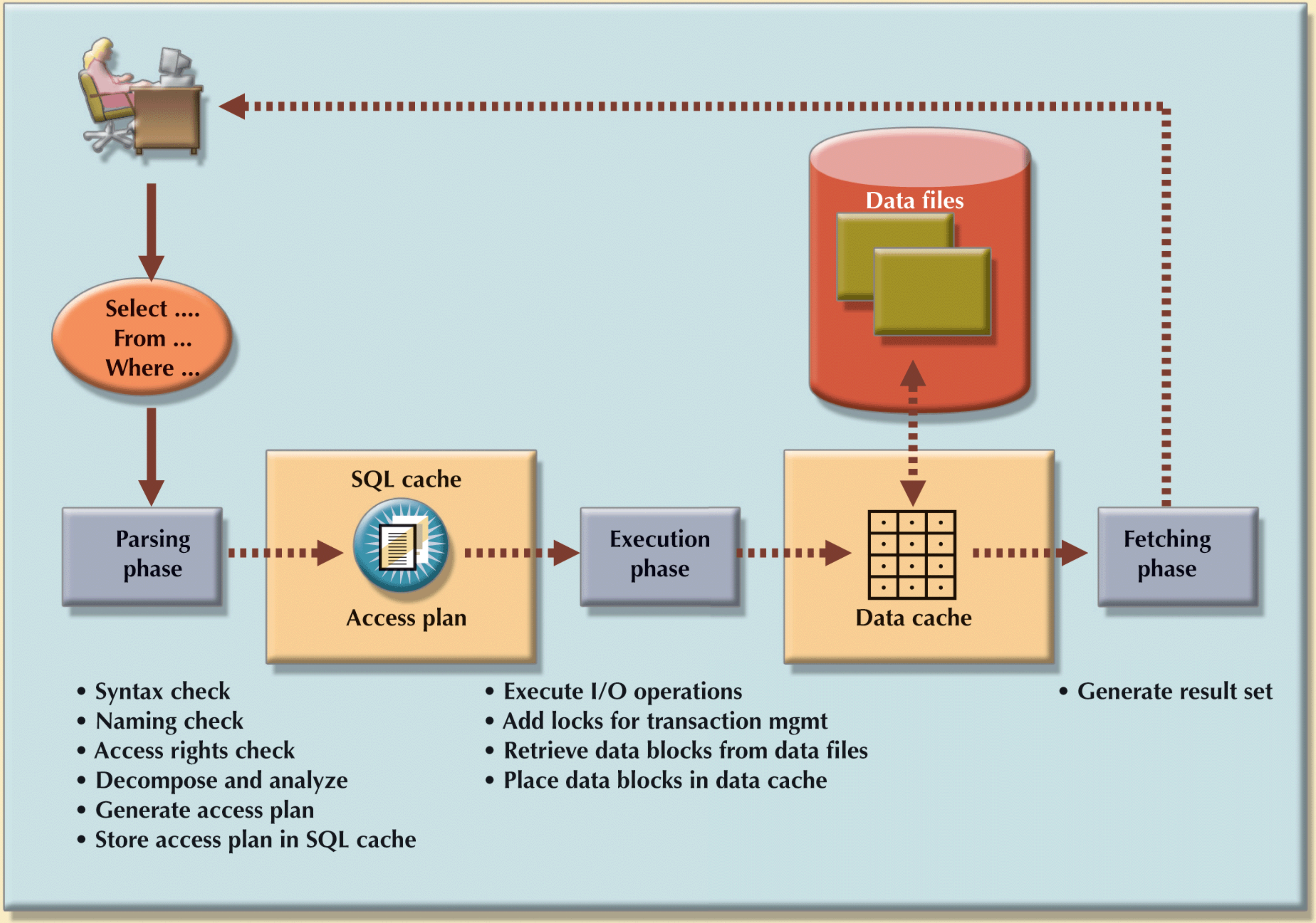
Posted by **CmdrTaco** on Thursday November 09, 2006 @11:45AM
from the this-is-never-good dept.

Last night we crossed over 16,777,216 comments in the database. The wise amongst you might note that this number is 2^24, or in MySQLese an unsigned mediumint. Unfortunately, like 5 years ago we changed our primary keys in the comment table to unsigned int (32 bits, or 4.1 billion) but neglected to change the index that handles parents. We're awesome! Fixing is a simple ALTER TABLE statement... but on a table that is 16 million rows long, our system will take 3+ hours to do it, during which time there can be no posting. So today, we're disabling threading and will enable it again later tonight. Sorry for the inconvenience. We shall flog ourselves appropriately. **Update: 11/10 12:52 GMT** by J : It's fixed.

Figure 11.1 - Basic DBMS Architecture



Cengage Learning © 2015



Select ....
From ...
Where ...

Parsing phase

- Syntax check
- Naming check
- Access rights check
- Decompose and analyze
- Generate access plan
- Store access plan in SQL cache

SQL cache

Access plan

Execution phase

- Execute I/O operations
- Add locks for transaction mgmt
- Retrieve data blocks from data files
- Place data blocks in data cache

Data files

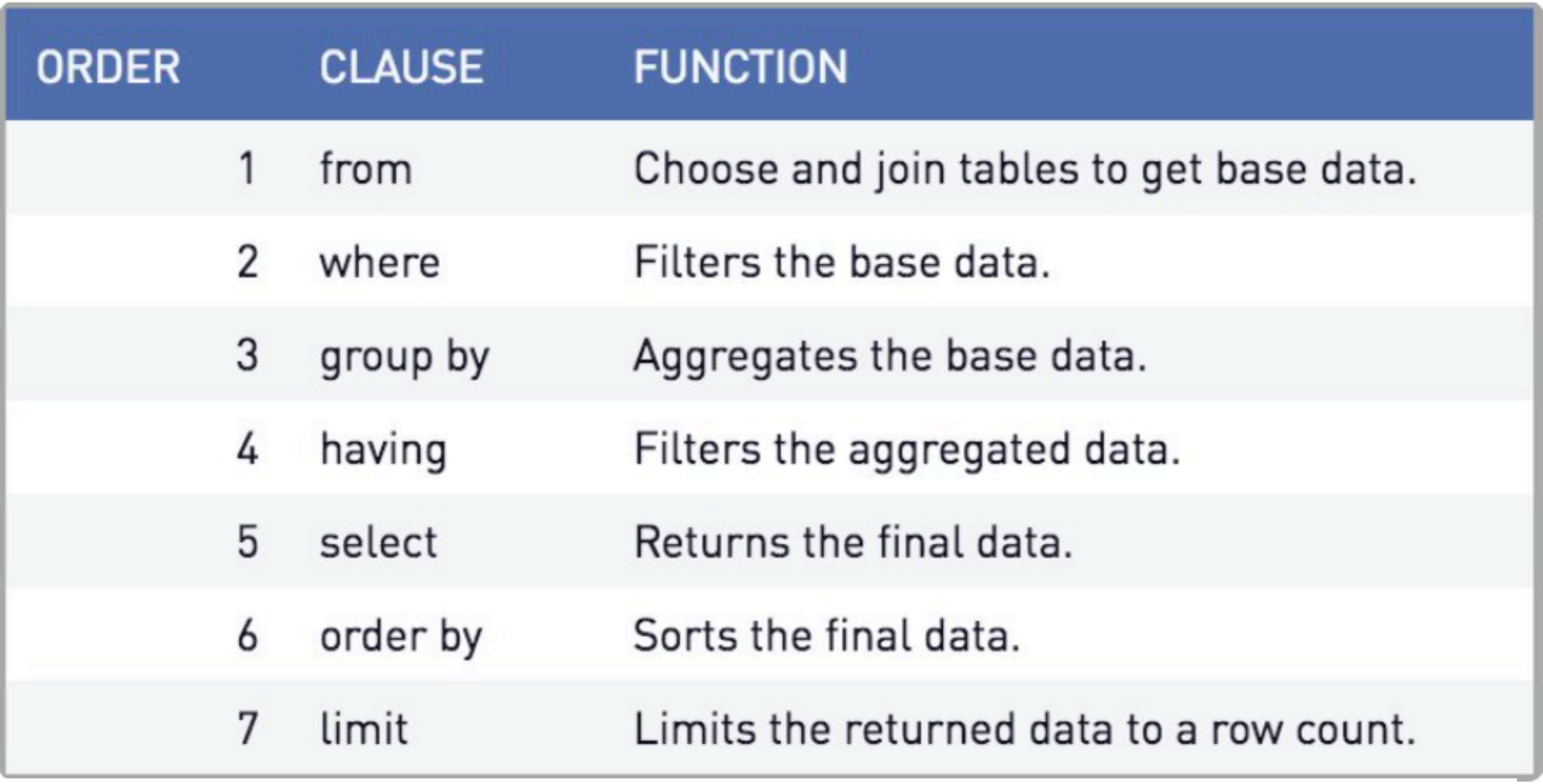
Data cache

Fetching phase

- Generate result set

Cengage Learning © 2015

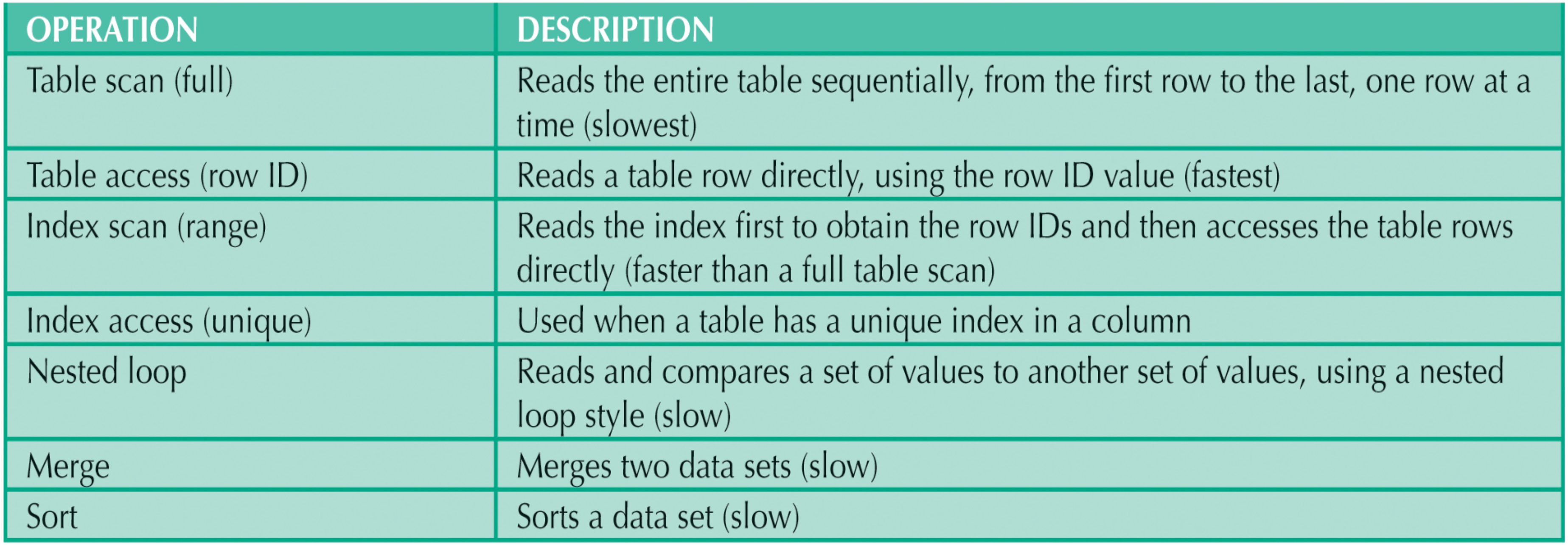
SQL Query Order of Execution*



ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.

* https://www.eversql.com/sql-order-of-operations-sql-query-order-of-execution/?

DB Access Plan I/O Ops



OPERATION	DESCRIPTION
Table scan (full)	Reads the entire table sequentially, from the first row to the last, one row at a time (slowest)
Table access (row ID)	Reads a table row directly, using the row ID value (fastest)
Index scan (range)	Reads the index first to obtain the row IDs and then accesses the table rows directly (faster than a full table scan)
Index access (unique)	Used when a table has a unique index in a column
Nested loop	Reads and compares a set of values to another set of values, using a nested loop style (slow)
Merge	Merges two data sets (slow)
Sort	Sorts a data set (slow)

Cengage Learning © 2015

Indexes

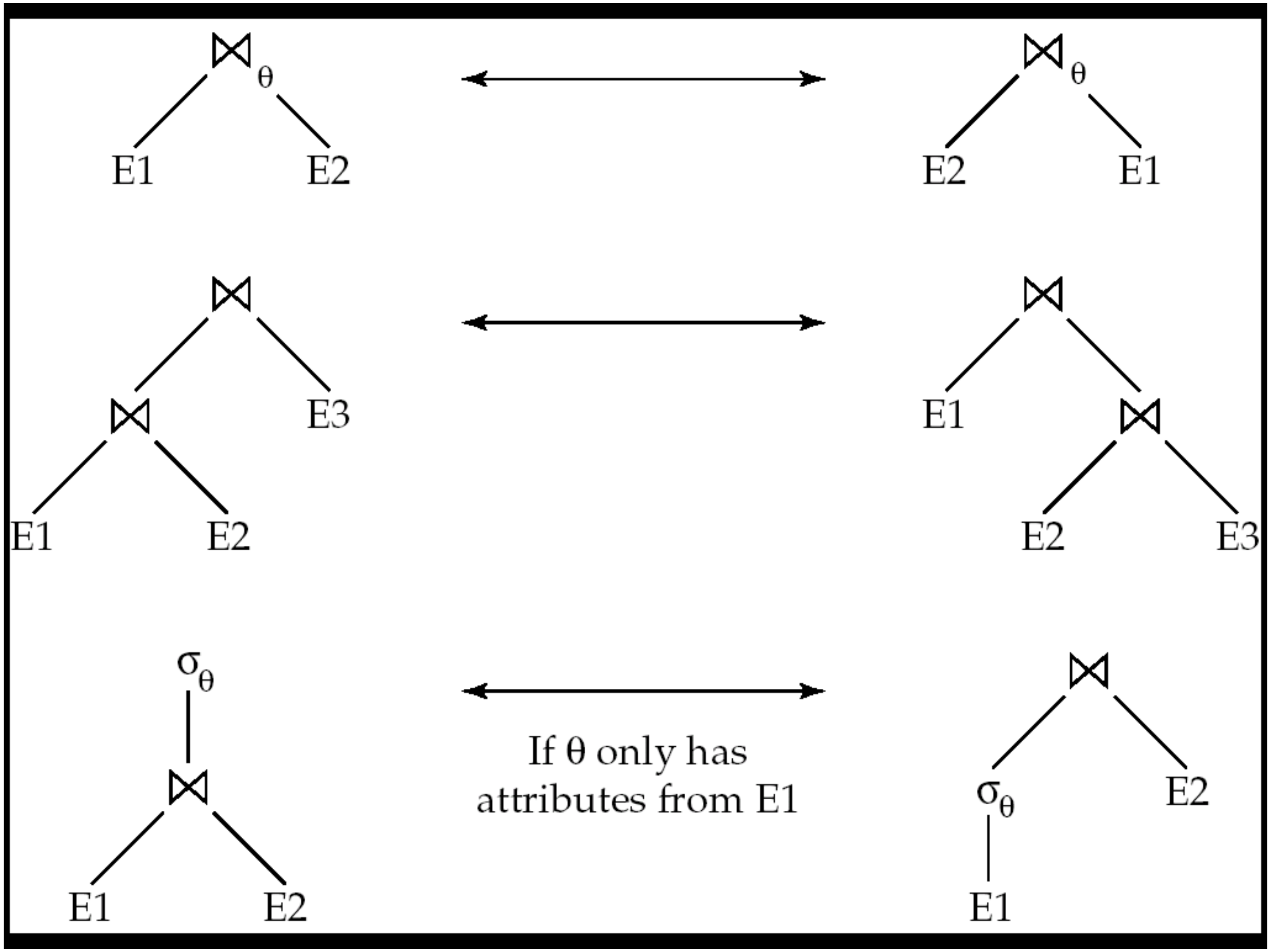
Indexes can allow duplicate values or not.

```
CREATE INDEX index_name
ON table_name (column1, column2, …);

CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, …);
```

You can DROP an index but can not ALTER one.

```
ALTER TABLE table_name
DROP INDEX index_name;
```

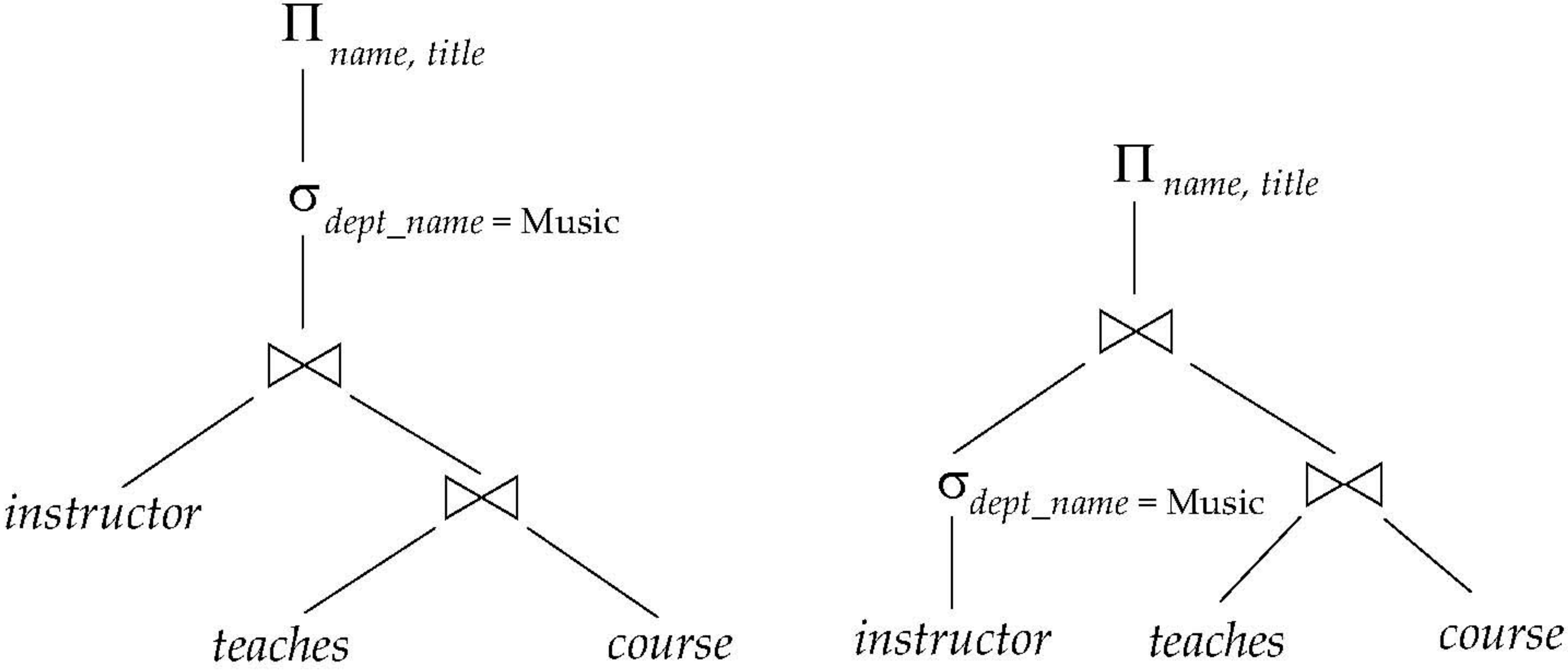
Some simple equivalencies



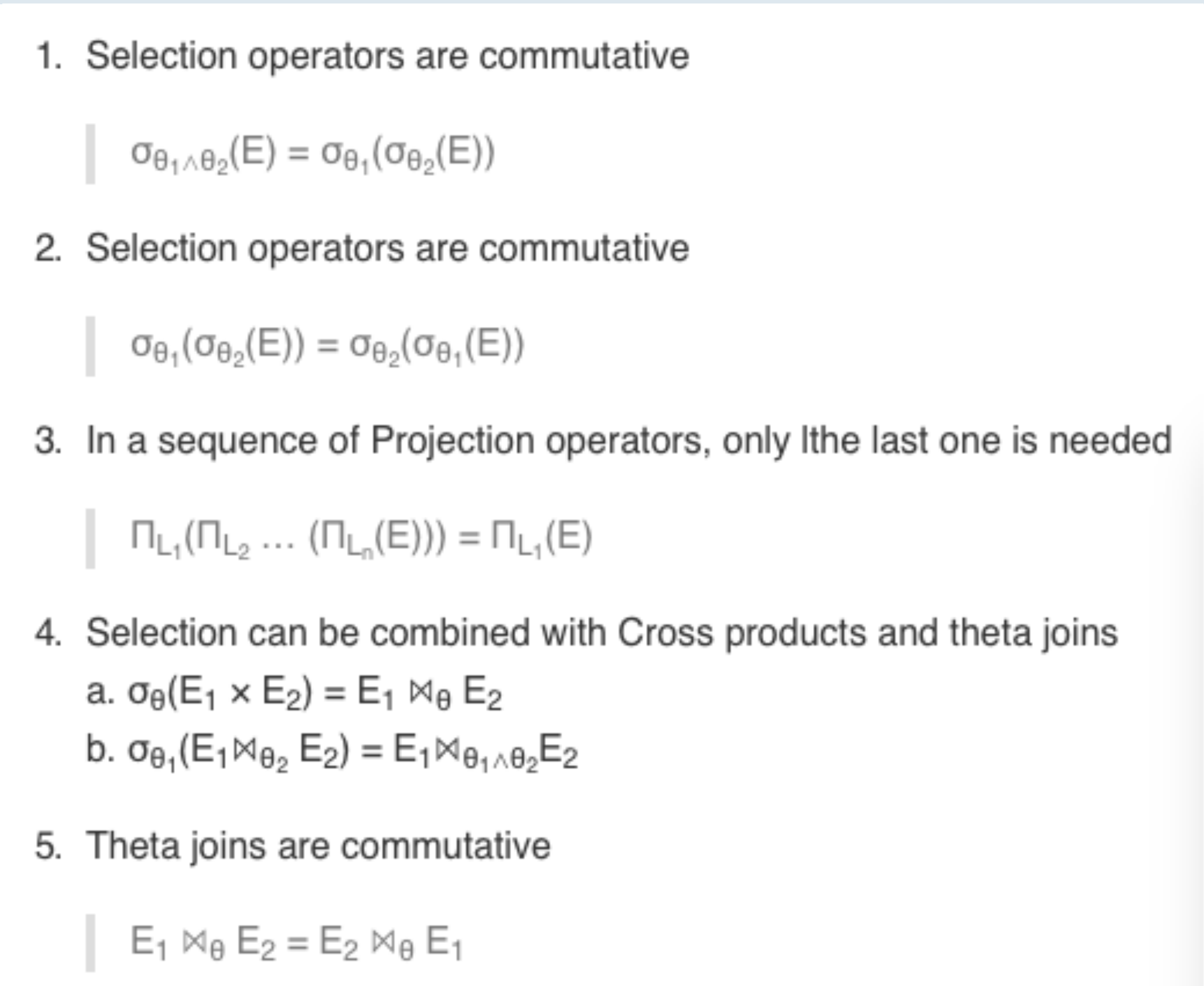
If θ only has attributes from E1

From Silberschatz, 11th ed.

Equivalent expressions



From Silberschatz, 11th ed.



1. Selection operators are commutative

> $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

2. Selection operators are commutative

> $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$

3. In a sequence of Projection operators, only lthe last one is needed

> $\Pi_{L_1}(\Pi_{L_2} \ldots (\Pi_{L_n}(E))) = \Pi_{L_1}(E)$

4. Selection can be combined with Cross products and theta joins
   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta joins are commutative

> $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$

6. Natural joins are a special case of theta joins, so they are also commutative
    a. $(E_1 \bowtie_{\theta} E_2) \bowtie E_3 = E_1 \bowtie_{\theta} (E_2 \bowtie E_3)$
    b. Theta joins are also associative sometimes:

> $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
>
> where θ_2 only involves attributes from E_2 and E_3 .

7. Selection distributes over theta-join sometimes:
    a. When θ_1 only involves attributes of E_1

> $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} E_2$

    b. When θ_1 only involves attributes of E_1 and θ_2 involves only attributes of E_2

> $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2))$

Example from Silberschatz

instructor(ID,pname,dept_name,salary)

teaches(ID, course_id, sec_id, semester, year)

course(course_id, title, dept_name,credits)

Find the names of all instructors in the Music department who taught in 2009 together with the course title of all the courses the instructors taught.

From Silberschatz, 11th ed.

$$\Pi_{pName,title}(\sigma_{dept\_name='Music' \cap year=2009}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$$

Apply 6A to transform this $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))$

into $(instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)$

to obtain

$$\Pi_{pName,title}(\sigma_{dept\_name='Music' \cap year=2009}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)))$$

From Silberschatz, 11th ed.

$$\Pi_{pName,title}(\sigma_{dept\_name='Music' \cap year=2009}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)))$$

Apply 7a to obtain this

$$\Pi_{pName,title}(\sigma_{dept\_name='Music' \cap year=2009}((instructor \bowtie teaches)) \bowtie \Pi_{course\_id,title}(course))$$

From Silberschatz, 11th ed.

$$\Pi_{pName,title}(\sigma_{dept\_name='Music' \cap year=2009}((instructor \bowtie teaches)) \bowtie \Pi_{course\_id,title}(course))$$

Apply 1 to break up the select

$$\sigma_{dept\_name='Music'}(\sigma_{year=2009}(instructor \bowtie teaches))$$

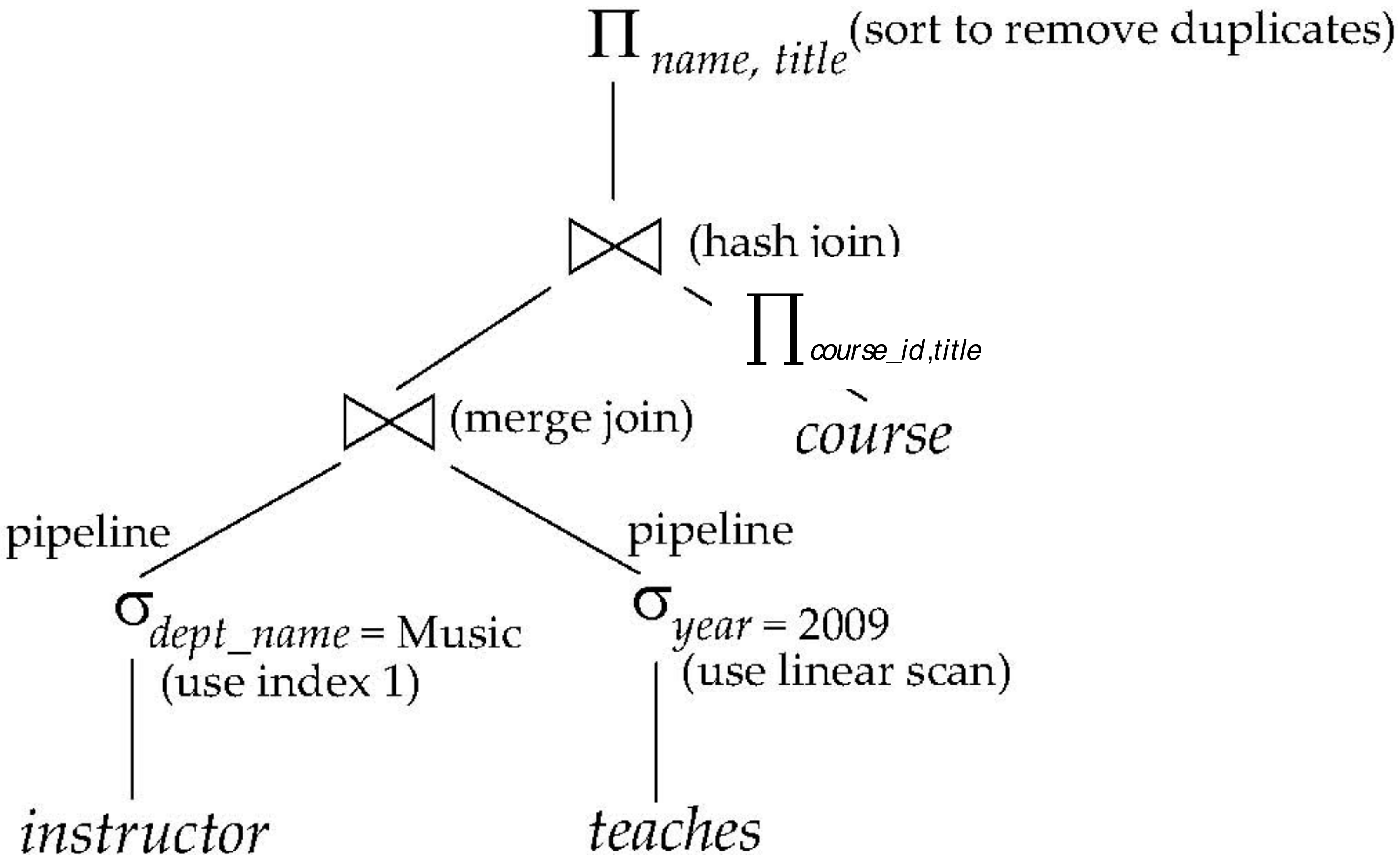
Apply 7a again

$$\sigma_{dept\_name='Music'} instructor \bowtie \sigma_{year=2009} instructor \bowtie teaches$$

Giving the final result

$$\Pi_{pName,title}(\sigma_{dept\_name='Music'} instructor \bowtie \sigma_{year=2009} instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)$$

From Silberschatz, 11th ed.



$\Pi_{name,\ title}$ (sort to remove duplicates)

$\bowtie$ (hash join)

$\Pi_{course\_id, title}$

course

$\bowtie$ (merge join)

pipeline

pipeline

$\sigma_{dept\_name = \text{Music}}$
(use index 1)

$\sigma_{year = 2009}$
(use linear scan)

instructor

teaches

From Silberschatz, 11th ed.

Evaluation Plan example

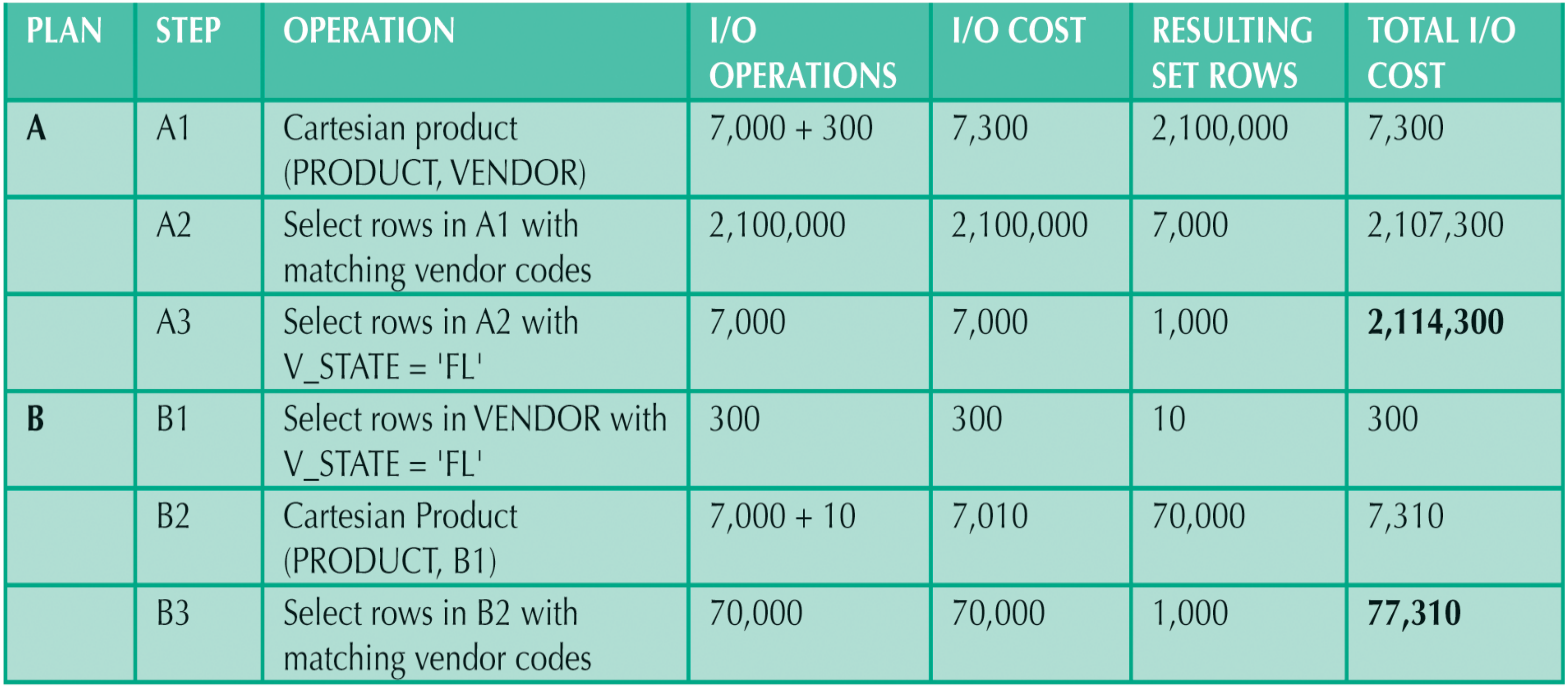
```
SELECT   P_Code, P_Descript, P_Price, V_Name, V_State
FROM     Product, Vendor
WHERE    Product.V_Code = Vendor.V_Code AND
         Vendor.C_State = ‘FL';
```

We know that:

1. the Product table has 7,000 rows
2. the Vendor table has 300 rows
3. 10 Vendors are in FL
4. 1,000 products come from the vendors in FL

Without doing a query, the optimizer only knows 1 & 2.

Access Plans vs. I/O Costs



PLAN	STEP	OPERATION	I/O OPERATIONS	I/O COST	RESULTING SET ROWS	TOTAL I/O COST
A	A1	Cartesian product (PRODUCT, VENDOR)	7,000 + 300	7,300	2,100,000	7,300
	A2	Select rows in A1 with matching vendor codes	2,100,000	2,100,000	7,000	2,107,300
	A3	Select rows in A2 with V_STATE = 'FL'	7,000	7,000	1,000	**2,114,300**
B	B1	Select rows in VENDOR with V_STATE = 'FL'	300	300	10	300
	B2	Cartesian Product (PRODUCT, B1)	7,000 + 10	7,010	70,000	7,310
	B3	Select rows in B2 with matching vendor codes	70,000	70,000	1,000	**77,310**

Cengage Learning © 2015

SQL query analysis tools

EXPLAIN ANALYZE is a profiling tool for your queries that will show you where MySQL spends time on your query and why. A great explanation is at

`https://dev.mysql.com/blog-archive/mysql-explain-analyze/`

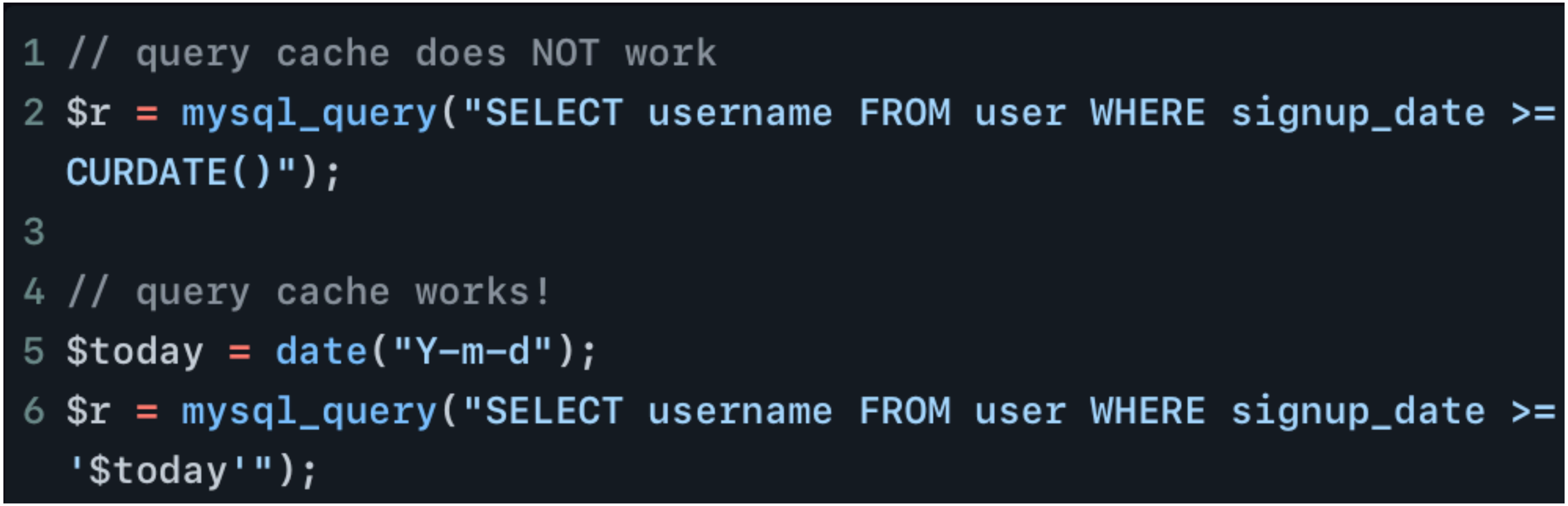
The MySQL optimizer determines the most efficient means of executing a query.
You can use Optimizer Tracing to see just how the query optimizer works

`https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_OPT_TRACE.html`

Writing efficient SQL

`Use the cache, Luke!`

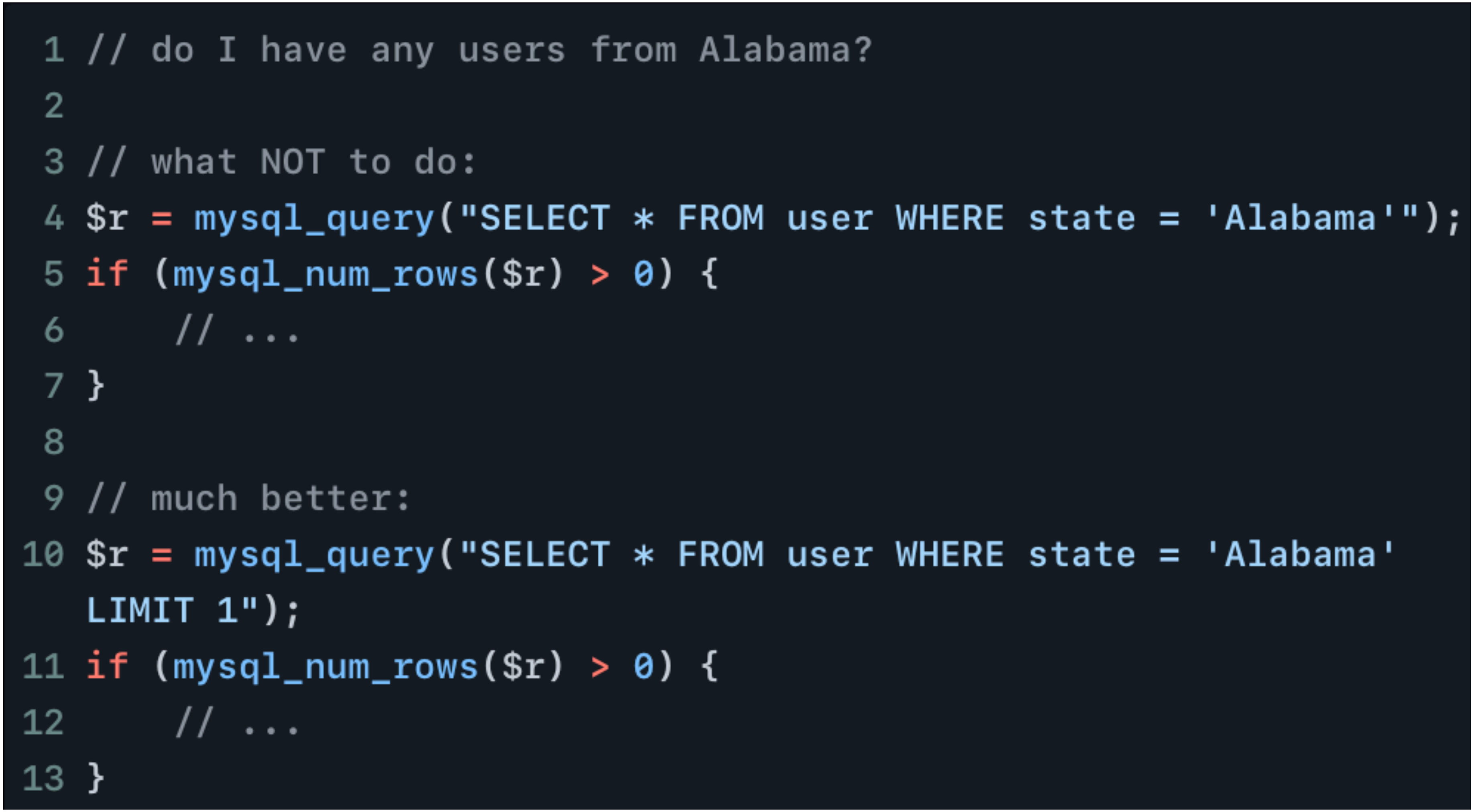
`Avoid non-determinism`



```
// query cache does NOT work
$r = mysql_query("SELECT username FROM user WHERE signup_date >= CURDATE()");

// query cache works!
$today = date("Y-m-d");
$r = mysql_query("SELECT username FROM user WHERE signup_date >= '$today'");
```

Writing efficient SQL

If you only want one, tell SQL!



```php
// do I have any users from Alabama?

// what NOT to do:
$r = mysql_query("SELECT * FROM user WHERE state = 'Alabama'");
if (mysql_num_rows($r) > 0) {
    // ...
}

// much better:
$r = mysql_query("SELECT * FROM user WHERE state = 'Alabama' LIMIT 1");
if (mysql_num_rows($r) > 0) {
    // ...
}
```

Writing efficient SQL

- Whenever possible (and enforceable) use CHAR(n) instead of VARCHAR(n) (and TEXT and BLOB) since fixed-size attributes are always faster.

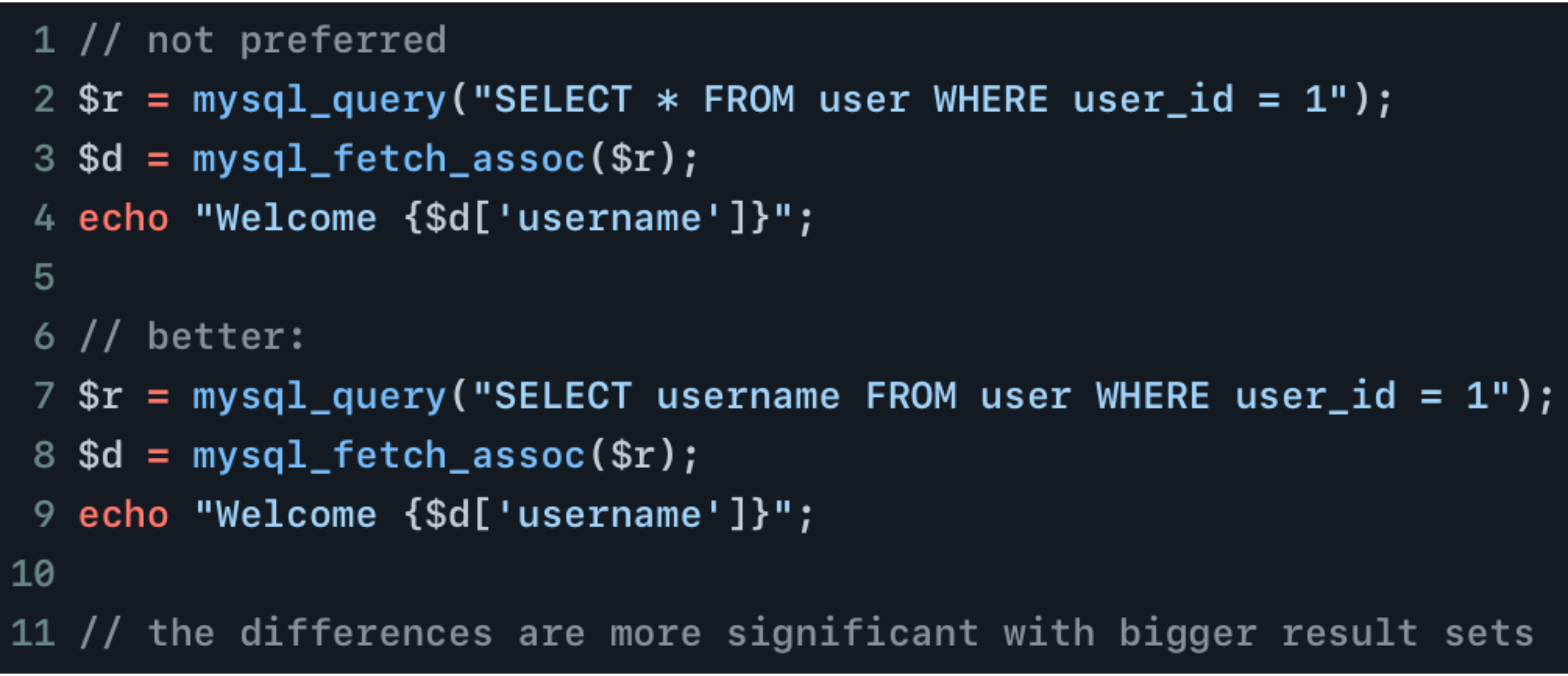
- Keep your primary keys integers whenever you can.

- Don't use DISTINCT when you have or could use GROUP BY.

- Avoid wildcard characters at the beginning of LIKE clauses. If the first characters are specified, then the DB can use the index to speed up the LIKE search/matching. The worst case is "%ski%" which prevents any index help

Writing efficient SQL

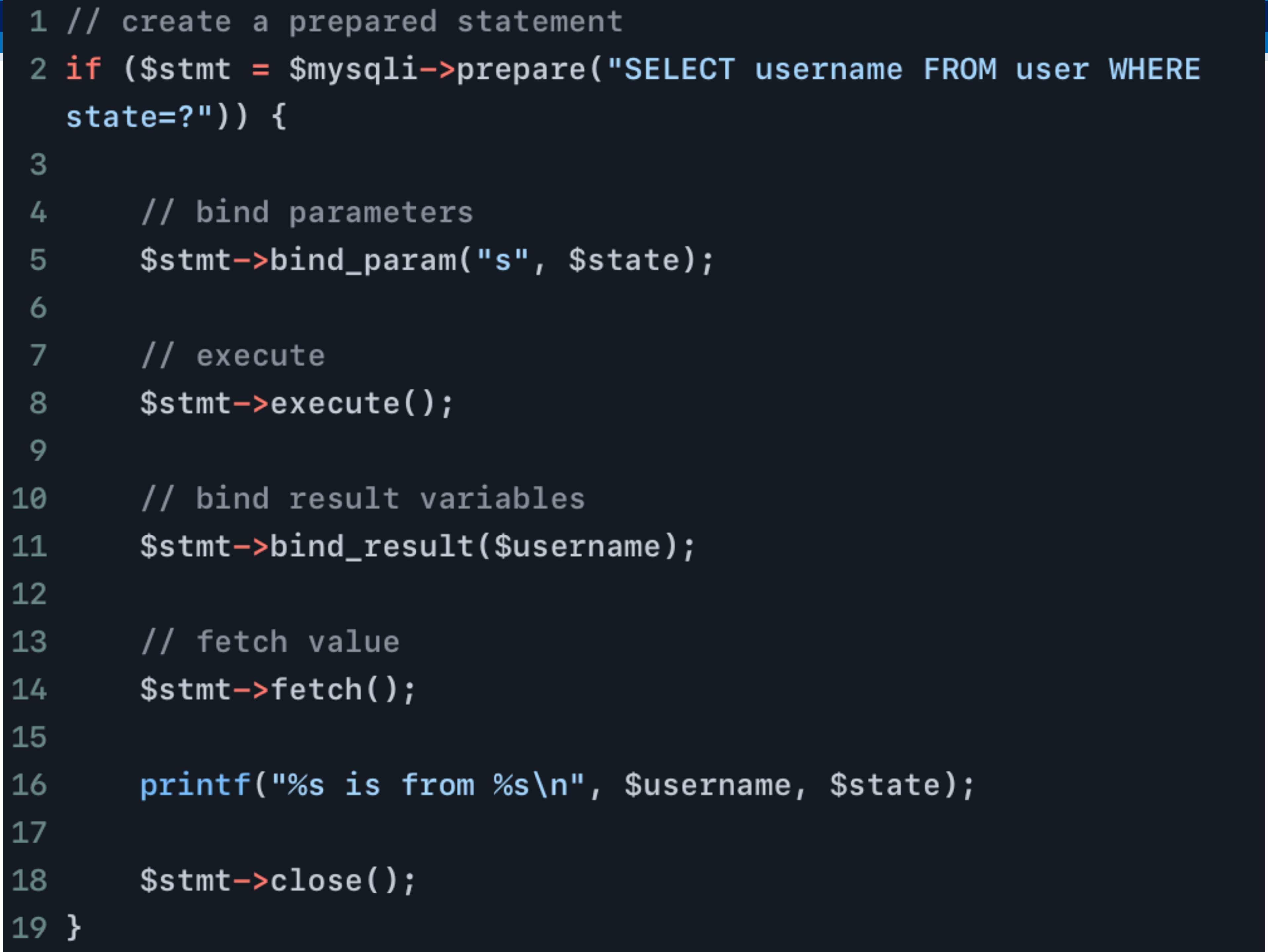
Avoid SELECT * when you can



```php
// not preferred
$r = mysql_query("SELECT * FROM user WHERE user_id = 1");
$d = mysql_fetch_assoc($r);
echo "Welcome {$d['username']}";

// better:
$r = mysql_query("SELECT username FROM user WHERE user_id = 1");
$d = mysql_fetch_assoc($r);
echo "Welcome {$d['username']}";

// the differences are more significant with bigger result sets
```

Writing efficient SQL

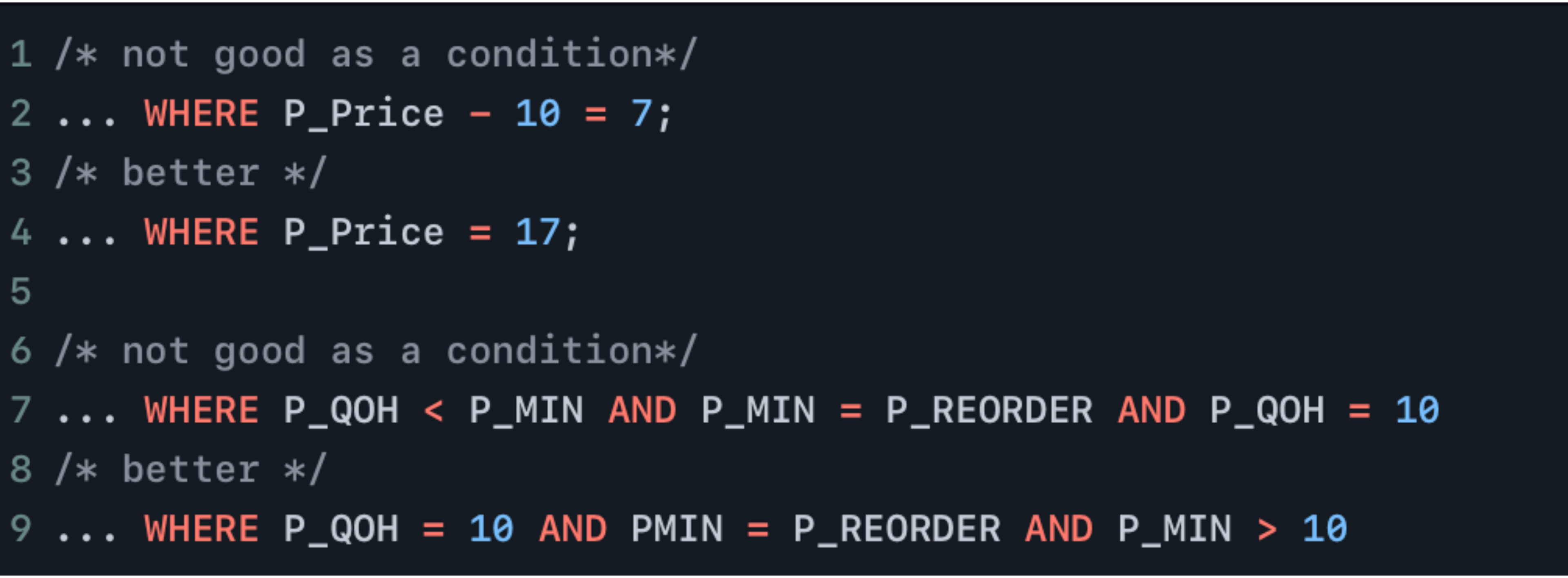
Use PREPARED STATEMENTS when getting user input



```php
// create a prepared statement
if ($stmt = $mysqli->prepare("SELECT username FROM user WHERE state=?")) {

    // bind parameters
    $stmt->bind_param("s", $state);

    // execute
    $stmt->execute();

    // bind result variables
    $stmt->bind_result($username);

    // fetch value
    $stmt->fetch();

    printf("%s is from %s\n", $username, $state);

    $stmt->close();
}
```

Writing efficient SQL

Use literals/constants in conditional expressions



```sql
/* not good as a condition*/
... WHERE P_Price - 10 = 7;
/* better */
... WHERE P_Price = 17;

/* not good as a condition*/
... WHERE P_QOH < P_MIN AND P_MIN = P_REORDER AND P_QOH = 10
/* better */
... WHERE P_QOH = 10 AND PMIN = P_REORDER AND P_MIN > 10
```

Writing efficient SQL

- When you know you will be joining two tables, make sure the attributes being used for that join are indexed!

- Always test equality conditions first - they're the easiest to process.

- Numeric field comparisons are always faster than character, date, and NULL comparisons.

- Functions are convenient, but using them in a conditional can be very expensive especially for larger tables.

- Avoid the use of the NOT logical operator when possible.

Writing efficient SQL

- When using multiple OR conditions, put the one most likely to be true first (this is a good thing to do in any programming language).
- Similarly, when using multiple AND conditions, put the one most likely to be false first (also a good thing to do in programming in general). [Coronel]
- Use the DESCRIBE to learn about tables and EXPLAIN to understand