The System (RDBMS)

1. **Appliance**
   The Database System as an Appliance (a review)
2. **Transaction Theory**
3. **Transaction Management**
4. **Other RDBMS Functions**
5. **Application Programming**

Version

initial	*2020-04-07*	Parke Godfrey
adapted	*2020-11-16*	Wenxiao Fu
last revised	*2020-11-17*	Parke Gdofrey

Acknowledgments

Thanks to

- *Jeffrey D. Ullman*
  for slides related to the textbook (*Stanford* slides_)
- *Jarek Szlichta*
  for significant refinements to the *Stanford slides*
- *Wenxiao Fu*
  for significant refinements

Printable version of talk

- Follow this link:
  The System [to pdf].
- Then *print* to get a **PDF**.

(*This only works correclty in **Chrome** or **Chromium***.)

1. Appliance

The Database System as an Appliance (a review)

The RDBMS as an Appliance

This course has been all about *how to use* an **RDBMS** — a *relational database management system* — as an *appliance* to help

- *manage* and
- *access*

our data.

Design (& Schema)

The *first* main section of the course was on *design*: how to map out a logical organization for the data we want to keep, called a *schema*.

- The **RDBMS** does not help us to do this design,
- but it can manage the *schema* and the data put into it for us.

The *relational model* does provide us a schema "language" that dictates *how* we can organize our data.

Queries (in SQL)

The *second* main section of the course was on *how* to access, and manipulate, the data in our database.

- The paradigm for this is to *query* over the database.
- The nearly universal query language for this — for relational database systems — is SQL.

What does the **RDBMS** provide?

- It does not help us to *compose* our SQL queries; only we know what we want.
- But it does evaluate our queries against the database — the schema and the data it contains — efficiently.

The RDBMS as a Blackbox

SQL, our way of interacting with our data, is *declarative*.

- We compose a query to state logically *what* we want, but we do not spell out a procedure for *how* this is to be accomplished.

  The **RDBMS** has to do the *how*.
- This is a *hallmark* of **RDBMS**'s, and what has made them so widely useful and used.

As such, in *this* course, we are *clients* of the **RDBMS**. We have to know how to work with it, but we do not have to know *how* it works.

We can treat the **RDBMS** as a *blackbox*.

What else should RDBMS clients (users) know?

What other functions of the **RDBMS** should users (**us**) master to use it effectively?

And that we should cover in this course?

Many! Too many to cover in a course of 12 to 15 weeks.

...but there are some important topics still to cover.

And, to be *expert* **RDBMS** users, do we need to look "under the hood" of the system at all?
Doesn't relational's *declarative* ideal mean we do not need to?!

Changing Data (*data in motion*)

We have mostly looked at the database as *static*, so far.

- We made the schema, then filled it with some data.
- We then obsessed on how we query over the data to retrieve the information we want.
- But data will be added and updated all the time!

So we also need a way to *manage* these *transactions*.

- *Integrity*: No transaction should be able to violate the schema's rules!
- *Concurrency*: Many transactions can be occuring "at the same time".

Concurrency

An **RDBMS** lets a database be shared by many, many clients at the same time.

- The *data* in the database is usually being added to, and modified, quite rapidly.
- And this is often being done *at the same time* by many clients.

Application Programming

and learning to live with others (concurrently)

As database users, we need to understand the consequences of "competing" with others for use of the database to be able to use the **RDBMS** effectively.

We need to know the *logic* of *how* the **RDBMS** deals with our "transactions".

> Oh, and to write sophisticated applications that use the **RDBMS** to manage the data, we cannot live with *just* SQL.
> How we can couple SQL with *programming languages* (e.g., **Java**) is needed.
> This is is called *application programming*.

Application Programming Interface(s)

Before we can dive into the “API” support **RDBMS**'s provide for application programming, we need to uderstand *how* the database system handles our transactions against the database.

This motivates the two key topics that we, as expert database users, need to master under this *third* section of the course:

1. transactions, *theory* & *management*
2. the application-programming interfaces

2. Transaction Theory

Important System Aspects

- For permanence, data is on disk.
- To work on data, it must be in main memory. (But main memory is volatile!)
- Main memory is orders of magnitude faster than secondary (“disk”) memory.

Primitive Operations

- *Read* a piece of data.
- *Write* a piece of data.

System-wise, a *transaction* is just a sequence of reads and writes.

Transactions

A *transaction* is a sequence of connected actions — *querying* and *updating* data — over the database.

The **RDBMS** ensures that our schema's rules are never violated.
It still needs to do this when there are transactions being processed at the same time (*concurrently*) and the transactions might logically conflict.

> *How* transactions can logically conflict and *what* guarantees the **RDBMS** can provide to prohibit possible logical conflicts is called *transaction theory*.
>
> And *how* the **RDBMS** performs this is called *transaction management*.

Transaction Theory

What can go “logically” wrong when two or more transactions “conflict” is *not* at all obvious.

What *can* go wrong was enumerated by **Jim Gray (@ Wikipedia)**, along with *what* transactional *properties* would prevent these issues and *how* to guarantee the properties (*transaction management*).

Dr. Gray won the *Turing Award* in 1998 for this work, “for seminal contributions to database and transaction processing research and technical leadership in system implementation.” (**Jim Gray (@ Wikipedia)**)

> Tragically, Dr. Gray vanished at sea piloting his 40-foot sailboat near San Francisco in *January 2007*.

Anomalies

vs the Transactional Properties

There are a number of logical *anomalies* that can arise when transactions "conflict".

This is akin to our study of *data anomalies* that could arise when a schema is not *normalized*.

A set of guiding principles — *transactional properties* — if adhered to will guarantee those anomalies cannot happen.

Again, analogous to how refining a schema to be in BCNF guarantees many types of *data anomalies* cannot occur within the schema.

> Which to study first? In this case, the *properties* give us good insight. So we will start with those. (Of course, Dr. Gray had to arrive at the properties from understanding the anomalies!)

The ACID Properties

- **A**: *atomicity*
- **C**: *consistency*
- **I**: *isolation*
- **D**: *durability*

These are the *four* properties we want the **RDBMS** to guarantee for us via its *transaction management*.

Each property says something about how any given transaction “sees” the database.
Recall, the data in the database is changing over time.

Atomicity

A transaction is a sequence of *actions* against the database (perhaps coming from an application program or a user with a “PostgreSQL” shell open).

Atomicity is also referred to as the *all-or-nothing* principle.

A number of the transaction's actions might make changes to the database. When the transaction is *done*, either *all* of the transaction's changes should now be part of the database (for any future transactions to see), or *none* should be (as if the transaction never existed)!

This is called *atomic* because it looks like the *transaction* was one large “action” that either happens or doesn't.

Atomicity: Commit *or* Rollback

Atomicity means that a transaction has one of two fates: *all* of its changes were *committed* to the database so now others can see the changes; or none were, as if the transaction *never* existed.

In the first case, we say the transaction *committed*.
In the second case, we say the transaction was *rolled back*.

In fact, there are two *special* actions a transaction can make: **commit** and **rollback**.
The database system *tries* to honour a **commit**, but cannot always. If it cannot, the transaction is *rolled back*.
And a transaction can *vanish* itself by calling **rollback**!
(The action **rollback** has the synonym **abort**.)

The Idea of Rollback

This notion that a transaction can be “undone” is very powerful!

We shall see it is a necessary consequence of **ACID**.

- We have to accept that our transaction might be *rolled back* by the database system at times, no fault of ours.
- But we can also “abort the mission” our transaction if we need to by *rolling back* ourselves.

Triggers & Rollback

Can a *trigger* be used as an *integrity constraint* to prohibit certain types of changes to be made?

Well, yes!

For example, let us say we want to prohibit that any employee's salary could be updated to a lower value.

- An *update* trigger can monitor for updates to *salary* in **Employee**. (the *event*)
- It checks whether the old salary is more than the new salary. (the *condition*)
- And what is its *action*? **rollback**!

The trigger's actions are within the scope of the transaction that *triggered* it.

Consistency

The **RDBMS** does not allow any change to the data that would violate the schema.

We call this *consistency*.

> Therefore, no transaction can ever *commit* that would leave the data of the database in a state that violates the schema's constraints.

This is something we are already quite used to.

Isolation

Transactions could not conflict if they were completed one at a time: all my transaction's changes are made and committed, then the next person's, etc.

Why not limit the **RDBMS** to handle *one* transaction at a time?

- It would be highly inefficient! A database is shared.
- You might have to wait a long, long time connecting to the database to do something.

Isolation: the illusion

So we need to let transactions run concurrently, for efficiency's sake.

But changes to the database should be *logically* equivalent to *some* order of the transactions happening as if one at a time, sequentially.

> We want to provide the *illusion* that the transactions are *isolated*, happening one at a time.

Transaction Schedules

A *schedule* is the *sequence* of actions executed by the database system.

- The server machine can only do one action at a time. (So no true *parallelism* here.)
- The actions from different transactions may be *interleaved* in the schedule (giving us *concurrency*).

If transactions's actions are not interleaved at all in the schedule, we call the schedule *serial*. This is *true* isolation.

When a schedule's results on the database are *equivalent* to that of some *serial* schedule, we call the schedule *serializable*.

Serializable schedules guarantee the property of *isolation*, the illusion of *true* isolation.

The Snapshot Principle

Because a transaction (X-act) sees the database as it looks as *after* the X-acts that *logically* came *before* it and sees *none* of the changes by X-acts that logically come *after* it (with respect to the equivalent serial schedule), it is as if the X-act sees the database as it is at *a single point in time* (the *snapshot*).

> *Isolation* is synonymous with the *snapshot* principle in the same way that *atomicity* is synonymous with the *all-or-nothing* principle.

Durability

Durability simply states that once a X-act has been *committed* by the database system, the changes made by the X-act are now visible by any X-act that comes after.

Commit means the changes are now permanently part of the data. (Well, at least until someone else comes along and changes the same data again!)

Serializable

We use the term *serializable* for the highest setting of *isolation level* for transactions. *Serializable* guarantees the **ACID** properties are never violated.

ACID — well, *atomicity* and *isolation* — guarantees that logical ***anomalies*** cannot occur, where we see "impossibilites" or *phantom* data that is not really there.

> There are steps down in *isolation level* that trade off allowing the possibility of some types of anomalies for increased concurrency. (Like dropping down in normal forms.)

Example: Interacting Processes

bar	beer	price
Joe's Bar	Bud	2.50
Joe's Bar	Miller	3.00

- Sally is querying **Sells** (above) for
  - Q1. the highest price, and
  - Q2. the lowest price Joe's Bar charges.
- Joe decides to
  - Q3. stop selling Bud and Miller (deletion), but
  - Q4. to sell only Heineken at $3.50 (insertion).

Sally's Program

Sally executes the following two SQL statements called (max) and (min) to help us remember what they do.

Q1. *(max)*

```
SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

Q2. *(min)*

```
SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

Joe's Program

At about the same time, Joe executes the following steps: *(del)* and *(ins)*.

Q3. *(del)*

```
DELETE FROM Sells
WHERE bar = 'Joe''s Bar';
```

Q4. *(ins)*

```
INSERT INTO Sells VALUES
('Joe''s Bar', 'Heineken', 3.50);
```

Interleaving of Statements

Except

- *(max)* must come before *(min)*, and
- *(del)* must come before *(ins)*,

there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions!

Example: Strange Interleaving

Suppose the steps execute in the order *(max)(del)(ins)(min)*.

What the highest price and lowest price does Sally get?

Joe's Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

Sally sees MAX < MIN!

Fixing the Problem by Using Transactions

If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.

She sees Joe's prices at some fixed time. Either

- before *(del)*,
- after *(del)* but before *(ins)*, or
- after *(ins)*,

but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

Sally	Joe
	(del)
	(ins)
(max)	
(min)	
commit	
	rollback

- If Sally executes her statements after *(ins)* but before the rollback,
- she sees a value, 3.50, that never existed in the database.

Solution

If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.

- If the transaction executes ROLLBACK instead, then its effects can never be seen.

Anomalies: Unrepeatable Reads / RW conflicts

Sally	Joe
(max)	
	(del)
	(ins)
	commit
(max)	

commit

- There are two different highest prices with the same beer!

SLIDEDECK in progress...

Application Programming

Any time a *program* connects to an **RDBMS** to issue SQL statements against a database, that is a *transaction*.

But how do we write programs that can “talk” to a database, “transact” with it?

> Most interaction with database systems is via programs, and not through direct SQL from a user.
>
> Fortuantely, how to *couple* programs to interact with a database is very standardized, just like SQL.

CLI (e.g., JDBC)

We need a *library* for our programming language that provided an API of calls so we can set up interaction with a database system

- This is generically called the *call library interface*)
- For Java, the “CLI” is called JDBC, for *Java-Database Connectivity*.
  (I will focus on JDBC here. But CLI's for other languages are fundamentally the same.)

The *only* way to interact with a database under an **RDBMS** is through SQL.

Therefore, JDBC must provide a way for a program to *send* SQL to the database server and to get the results back.

The JDBC Exchange

1. *sql-str*: In the **Java** program, we can store or build an SQL statement in a string.
2. *conn*: A call establishes a connection to a database server for a given database.
3. *statement*: Using the *conn* object, the *sql-str* is *prepared*. This sends the SQL to the database server, which parses it and "prepares" to execute the SQL commands when called.
4. *result-set*: calling *execute* on *statement* inkoves the database server to run the SQL statement and return the results.

The Result Set

Let us consider an SQL query that was *prepared* and *executed*. How does the **Java** program process the results? A query returns a *set* — okay, a *multi-set* — of tuples.

It is done via a specific type of *iterator*, a *cursor* (class `ResultSet` in JDBC). With the iterator, the program can iterate through the tuples of the answer set.

The Cursor

On the database-side of the fence, a *cursor* is an open *transaction*. And the snapshot semantics is ensured.

See ***JDBC: A small tutorial*** **[pdf]** to see the concrete steps.

And see **JDBC: Example** for an example.

Conclusion

And that concludes this review of *The System* for *transactions* and *applications*.

Study well!

Keep well!

The System | EECS-3421A: *Introduction to Database Systems* | Fall 2020