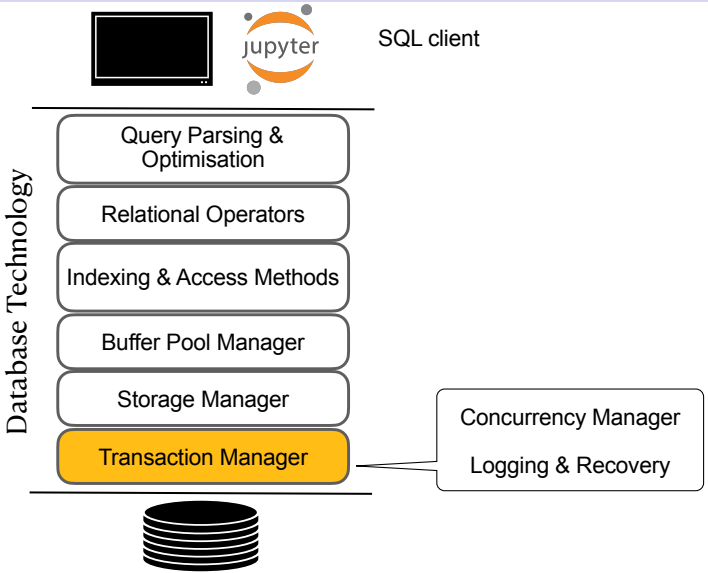
COL362/632 Introduction to Database Management Systems

Transactions

Kaustubh Beedkar

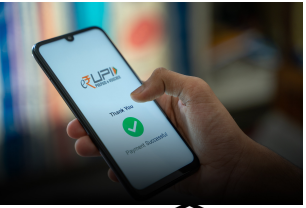
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Course Overview



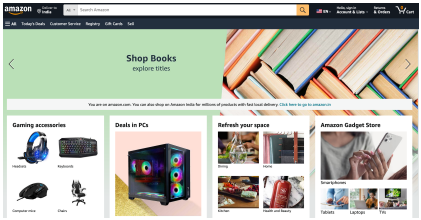
Transactions

- Major component of database systems
- Critical for many applications



IRCTC's payment gateway gains traction, handles 125,000 transactions daily

IRCTC introduces Autopay feature to ensure more reliable ticket booking



Cross shard transactions at 10 million requests per second

// By Daniel Tahara · Nov 09, 2018

Dropbox stores petabytes of metadata to support user-facing features and to power our production infrastructure. The primary system we use to store this metadata is named Edgestore and is described in a previous blog post, (Re)introducing Edgestore. In simple terms, Edgestore is a service and abstraction over thousands of MySQL nodes that provides users with strongly consistent, transactional reads and writes at low latency.

Jim Gray

WIKIPEDIA
The Free Encyclopedia

Create account Log in

Jim Gray (computer scientist)

30 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

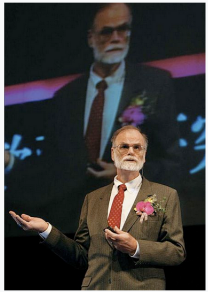
James Nicholas Gray (1944 – declared dead in absentia 2012) was an American computer scientist who received the Turing Award in 1998 "for seminal contributions to database and transaction processing research and technical leadership in system implementation".[4]

Early years and personal life [edit]

Gray was born in San Francisco, the second child of Ann Emma Sanbrailo, a teacher, and James Able Gray, who was in the U.S. Army; the family moved to Rome, Italy, where Gray spent most of the first three years of his life; he learned to speak Italian before English. The family then moved to Virginia, spending about four years there, until Gray's parents divorced, after which he returned to San Francisco with his mother. His father, an amateur inventor, patented a design for a ribbon cartridge for typewriters that earned him a substantial royalty stream.[2]

After being turned down for the Air Force Academy he entered the University of California, Berkeley as a freshman in 1961. To help pay for college, he worked as a co-op for General Dynamics, where he learned to use a Monroe calculator. Discouraged by his chemistry grades, he left Berkeley for six months, returning after an experience in industry he later described as "dreadful".[2] Gray earned his B.S. in engineering mathematics (Math and

Jim Gray



Gray in 2006

Born James Nicholas Gray

Transactions

- A transaction is a unit of work comprising sequence of updates with the **all-or-nothing** property
- Example transaction
  - read(A)
  - A := A - 50
  - write(A)
  - read(B)
  - B := B + 50
  - write(B)
- Two key challenges
  1. Failures (e.g., hardware, software, network, etc.)
  2. Concurrency (multiple transactions at the same time)

ACID Properties

Atomicity

Either all or nothing: The database must ensure that updates of a partially executed transaction are not reflected in the database

Consistency

A transaction as a whole must not violate integrity constraints

Isolation

Transactions must appear to execute one after the other in sequence

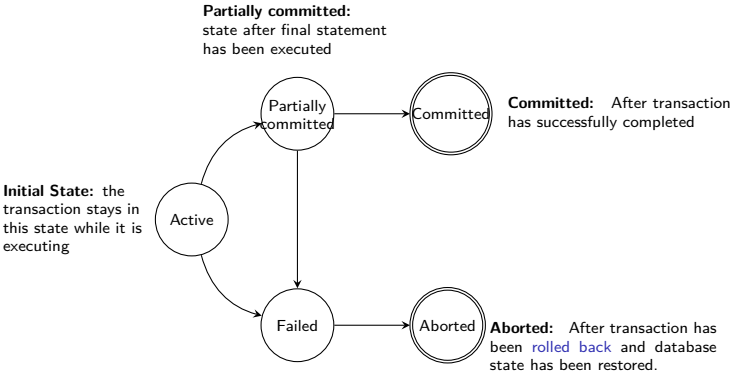
Durabaility

Updates to the database must persist after transaction is completed, even if there are failures

Example transaction

1. read(A)
2. A := A - 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Transaction States



ACID Properties

Why is hard to provide ACID properties?

- Concurrent transactions
  - Improved throughput & resource utilization
  - Reduced waiting time (low latency)
  - But, Isolation problems!
- Failures
  - Can happen at anytime
  - Atomicity and Durability problems!

Schedules

A **schedule** is a sequence of instructions that specify the chronological order in which instructions are executed.

Serial Schedule

Examples

A serial schedule: T_1 is followed by T_2

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

A serial schedule: T_2 is followed by T_1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Concurrent Schedule

A **serializable** schedule

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

A **non-serializable** schedule

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Serializable Schedules

- Database must ensure that the schedule is serializable
- Different notions
  - Conflict serializability
  - View serializability
- Simplified view
  - Never commute individual instructions
  - Only care about reads and writes

Conflict Serializability

Conflicts

Two instructions conflict iff both access the same data item and at least one of them wrote the data item

- Read-Write
- Write-Read
- Write-Write

Conflicts enforces a temporal order!

Definition (Conflict Serializable)

A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting instructions

Conflict Serializability (Example)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Conflict Serializability

Testing for conflict serializability

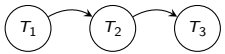
- Precedence graph
  - A graph with a node for each transaction T_i, and
  - an edge $T_i \rightarrow T_j$ whenever an instruction I_i in T_i conflicts with, and comes before an instruction I_j in T_j

- A schedule is serializable iff the precedence graph is acyclic

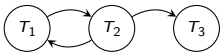
Conflict Serializability

Testing for conflict serializability Q: which of the following schedules is conflict serializable?

T_1	T_2	T_3
	read(A)	
read(B)		
	write(A)	
		read(A)
write(B)		
		write(A)
	read(B)	
	write(B)	



T_1	T_2	T_3
	read(A)	
read(B)		
	write(A)	
	read(B)	
		read(A)
write(B)		
		write(A)
	write(B)	



Dealing with Failures

What could go wrong here?

T_1	T_2
read(A)	
write(A)	
	read(A)
	commit
read(B)	
abort	

- Cannot abort T_1 because we cannot undo T_2
- This schedule is **not recoverable**

Recoverable Schedules

- A schedule is **recoverable** if
  - it is conflict serializable, and
  - T_j reads data item written by T_i, then commit of T_i must appear before commit of T_j

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
commit	
	commit

- If T_1 aborts, then T_2 also aborts

Cascading Aborts

- If a transaction T aborts, abort all transactions T' that have read data items written by T
- Can lead to the undoing of significant amount of work

T_1	T_2	T_3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

- If T_1 fails, then T_2 and T_3 must also be rolled back
- It is desirable to have schedules that are **cascadeless**
  - Whenever a transaction T reads a data item written by transaction T', then commit of T' must happen before read of T

Recap of Schedules

Serializability

- Serial
- Serializable
- Conflict or view serializable

Recoverability

- Recoverable
- Cascadeless

Recap of Schedules