Databases and Advanced Data Techniques

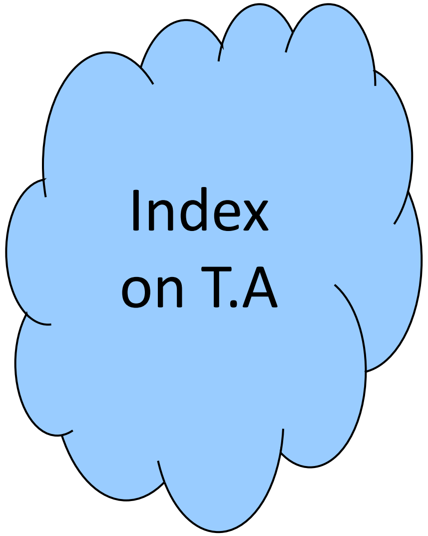
Tarapong Sreenuch

Indxes and Transactions

Indexes

- Primary mechanism to get improved performance on a database
- Persistent data structure, stored in a database
- Many interesting implementation techniques
  - In this course, we will only be focusing on user/application perspective.

Functionality

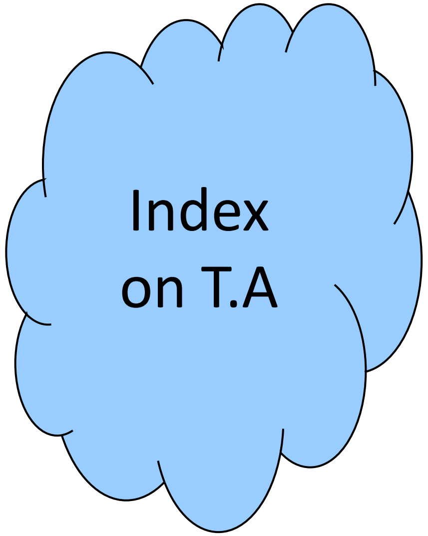


T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Functionality

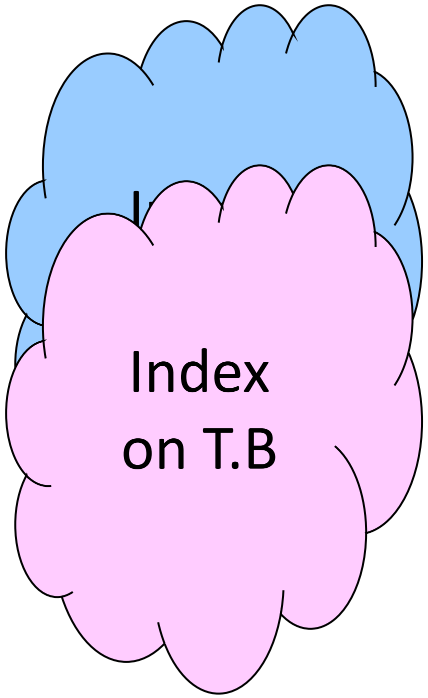
T.A = 'cat'



T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Functionality

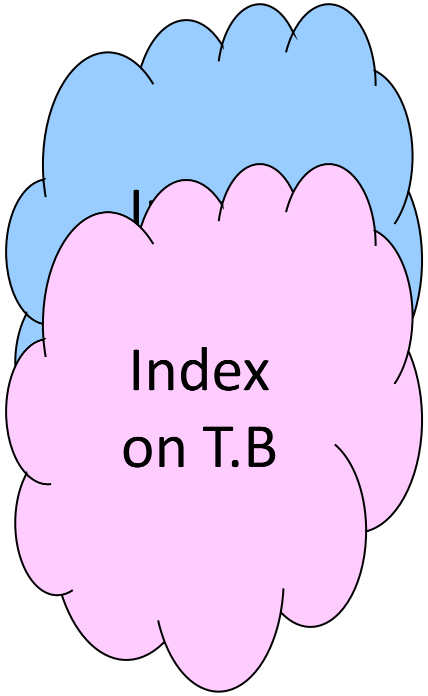


T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Functionality

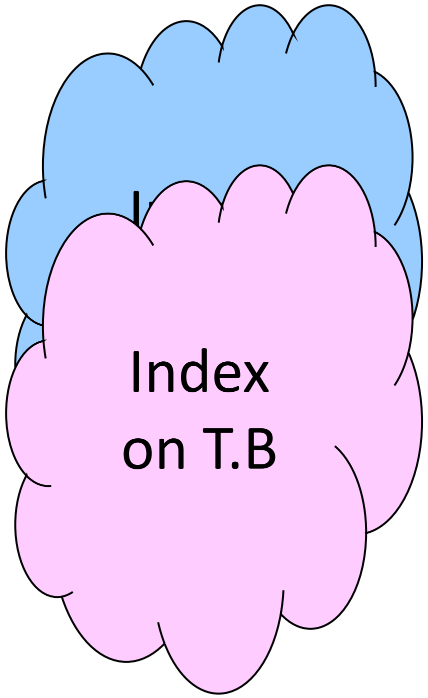
$T.B = 9$



T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

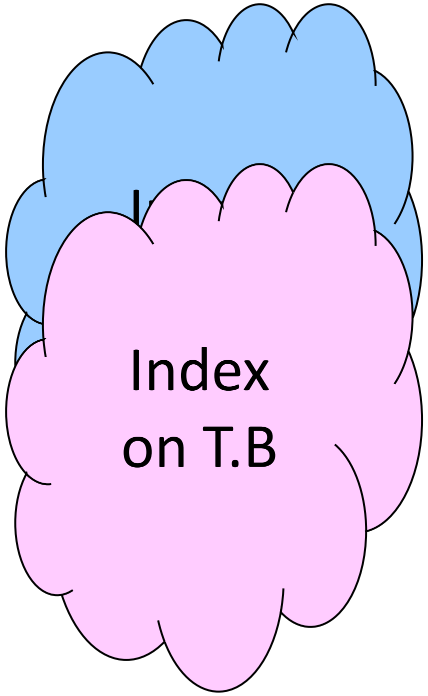
Functionality



T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

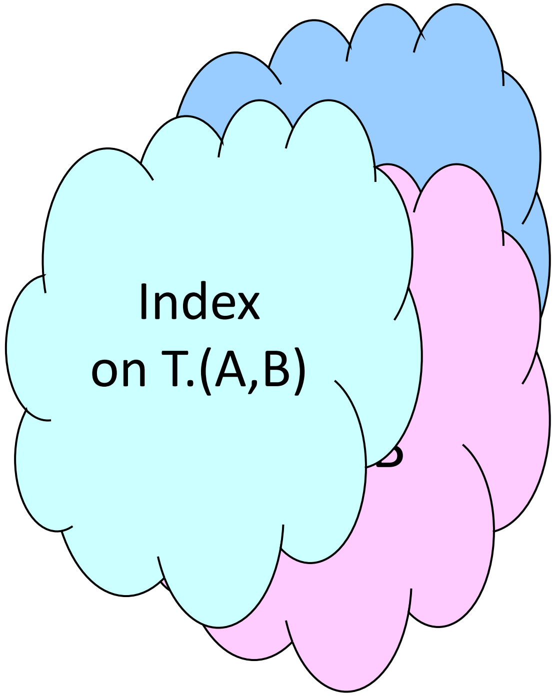
Functionality



T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Functionality



T.A = 'cat' AND T.B = 8

T

	A	B	C
1	cat	2	…
2	dog	5	…
3	cow	1	…
4	dog	9	…
5	cat	2	…
6	cat	8	…
7	cow	6	…
	…	…	…

Utility

- Index = difference between full table scans and immediate location of tuples
  - Order of magnitude performance difference

- Underlying data structures
  - Balanced trees (B trees, B+ trees)
  - Hash tables

Index Operation

```
SELECT sName
FROM Student
WHERE sID = 18942
```

Index on sID

Many DMBS build indexes automatically on PRIMARY KEY (and sometimes UNIQUE) attributes.

Index Search Operation

```
SELECT sID
FROM Student
WHERE sName = 'Mary' AND GPA > 3.9
```

Index on sName ← Hash- or Tree based indexes are required in order to support equality search.

Index on GPA ← Tree based indexes are required in order to support inequality search.

Index on (sName, GPA)

Index Search Operation

```
SELECT sName, cName
FROM Student, Apply
WHERE Student.sID = Apply.sID
```

Index

Index

Query Planning & Optimisation

Downsides of Indexes

Benefit of an index depends on:

- Extra Space ← *Marginal Downside*
- Index creation ← *Medium Downside*
- Index Maintenance ← *Cost Benefit Tradeoffs*

Which Indexes To Create?

Benefit of an index depends on:

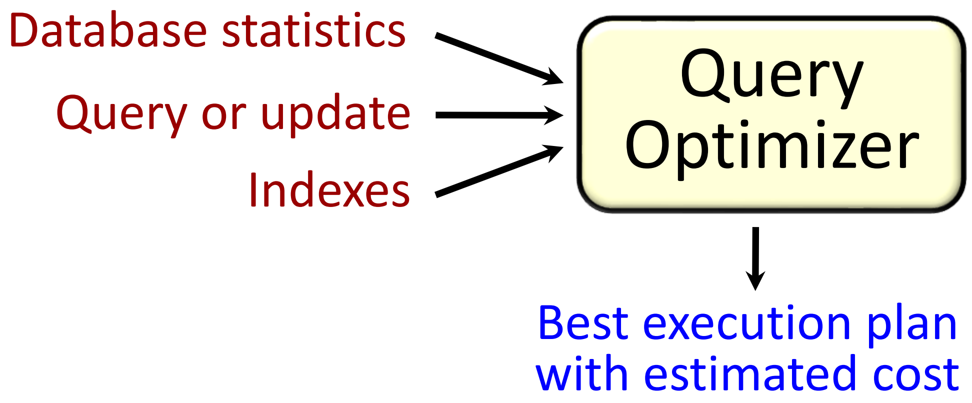
- Size of table (and possibly layout)
- Data distributions
- Query vs. update load

Physical Design Advisors

Input: Database (Statistics) and Workload

Output: Recommendation Indexes

We're looking for Indexes that their benefits outweigh the drawbacks.



SQL Syntax

```
CREATE INDEX IndexName ON T(A)
CREATE INDEX IndexName ON T(A1,A2,…,An)
CREATE UNIQUE INDEX IndexName ON T(A)
DROP INDEX IndexName
```

Recap: Indexes

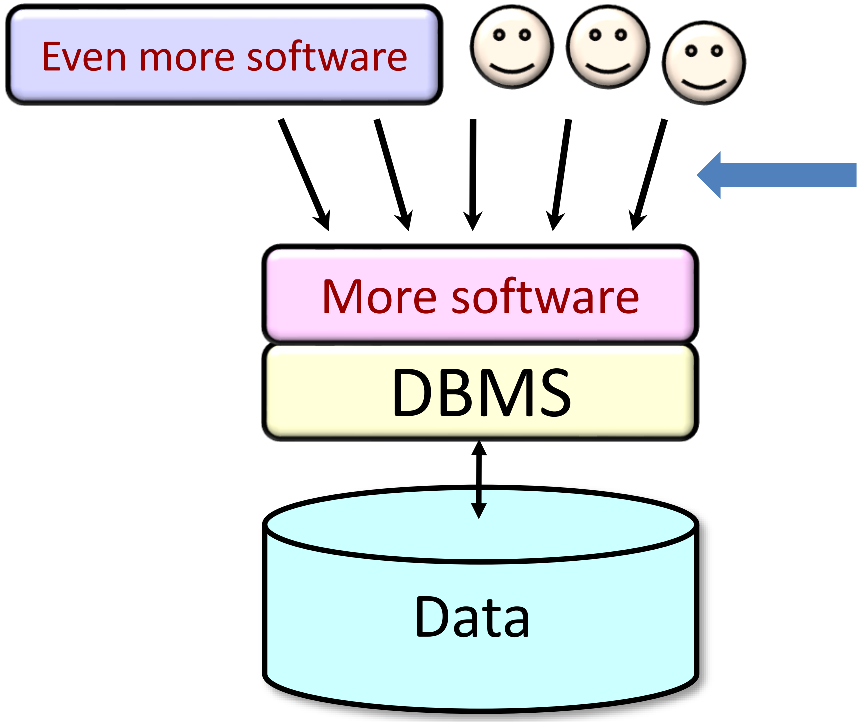
- Primary mechanism to get improved performance on a database
- Persistent data structure, stored in database
- Ways to implementation it:
    - Hash, B-Tree, etc.

Transactions

Motivated by two independent requirements

- Concurrent database access
- Resilience to system failures

Concurrent Database Access



Select…
Update…
Create Table…
Drop Index…
Help…
Delete…

Concurrent Access: Attribute-Level Inconsistency

```
UPDATE College SET enrollment = enrollment + 1000
WHERE cName = ‘University of London’
```

… concurrent with …

```
UPDATE College SET enrollment = enrollment + 1500
WHERE cName = ‘University of London’
```

Concurrent Access: Tuple-Level Inconsistency

```
UPDATE Apply SET major = 'CS'
WHERE sID = 123
```

… concurrent with …

```
UPDATE Apply SET decision = 'Y'
WHERE sID = 123
```

Concurrent Access: Table-Level Inconsistency

```
UPDATE Apply SET decision = 'Y'
WHERE sID IN ( SELECT sID
               FROM Student
               WHERE GPA > 3.9 )
```

… concurrent with …

```
UPDATE Student SET GPA = (1.1)*GPA
WHERE sizeHS > 2500
```

Concurrent Access: Multi-Statement Inconsistency

```
INSERT INTO Archive
  SELECT * FROM Apply WHERE decision = ‘N’;
DELETE FROM Apply WHERE decision = ‘N’;
```

… concurrent with …

```
SELECT COUNT(*) FROM Apply;
SELECT COUNT(*) FROM Archive;
```

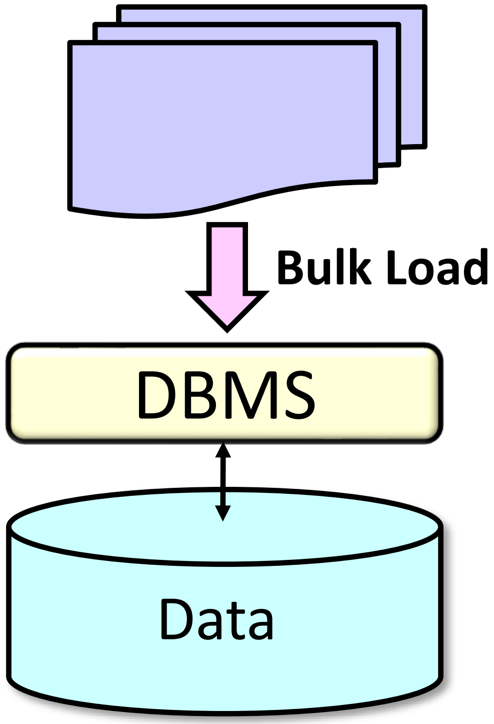
Concurrent Goal

Execute sequence of SQL statements so they appear to be running in isolation

- A simple solution is to execute them in isolation.

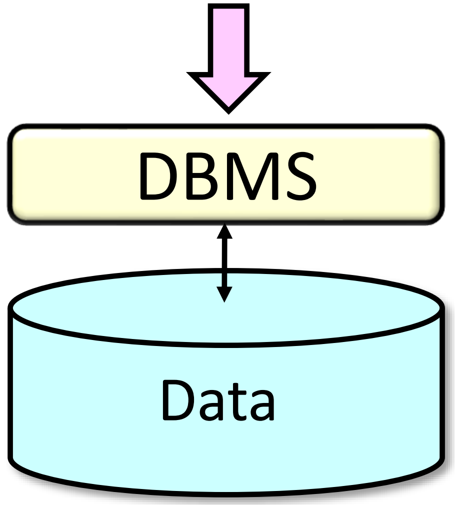
But still we want to enable concurrency whenever safe to do so.

Resilience to System Failures

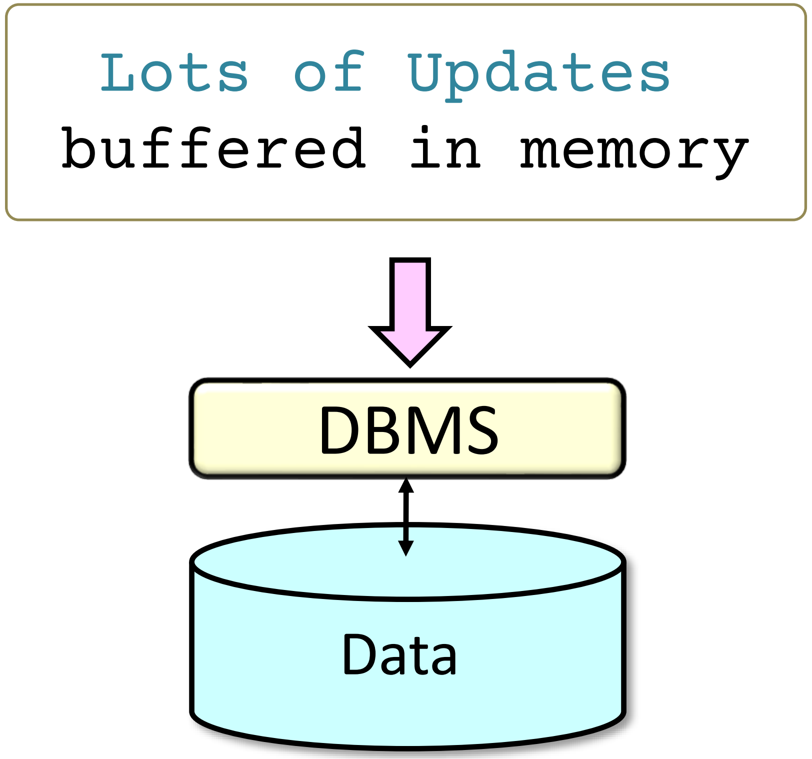


Resilience to System Failures

```
INSERT INTO Archive
   SELECT * FROM Apply WHERE decision = 'N';
DELETE FROM Apply WHERE decision = 'N';
```

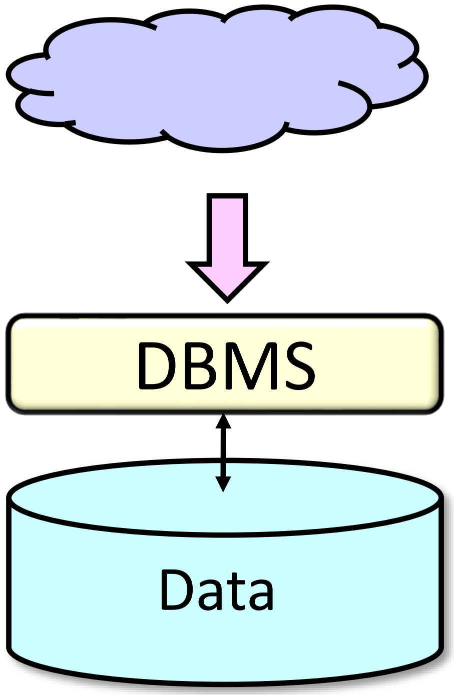


Resilience to System Failures



System-Failure Goal

Guarantee all-or-nothing execution, regardless of failures



Solution for both Concurrency and Failures

TRANSCATIONS

Guarantee all-or-nothing execution, regardless of failures

- Transactions appear to run in isolation
- If the system fails, each transaction's changes are reflected either entirely or not at all.

TRANSCATIONS

A transaction is a sequence of one or more SQL operations treated as a unit.

SQL Standard:

- Transaction begins automatically on first SQL statement.
- On 'COMMIT' transaction ends and new one begins.
- Current transaction ends on session termination
- 'AUTOCOMMIT' turns each statement into transaction.

Solution for both Concurrency and Failures

TRANSCATIONS

A transaction is a sequence of one or more SQL operations treated as a unit.

ACID Properties

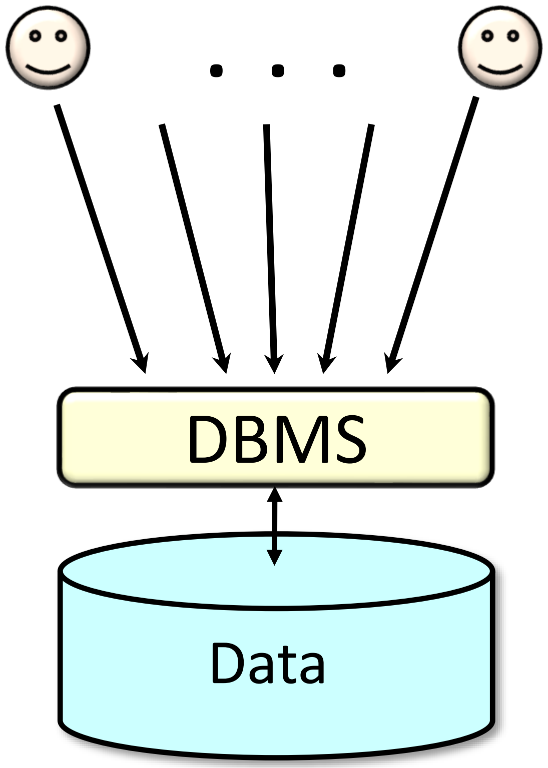
Atomicity

Consistency

Isolation

Durability

ACID Properties: Isolation



Serialisability

Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions.

Locking

Concurrent Access: Attribute-Level Inconsistency

T1

```
UPDATE College SET enrollment = enrollment + 1000
WHERE cName = 'University of London'
```

… concurrent with …

T2

```
UPDATE College SET enrollment = enrollment + 1500
WHERE cName = 'University of London'
```

T1; T2
or
T2; T1

…order doesn't matter…

Concurrent Access: Tuple-Level Inconsistency

```
UPDATE Apply SET major = 'CS'
WHERE sID = 123
```

… concurrent with …

```
UPDATE Apply SET decision = 'Y'
WHERE sID = 123
```

T1; T2
or
T2; T1

…order doesn't matter…

Concurrent Access: Table-Level Inconsistency

T1

```
UPDATE Apply SET decision = 'Y'
WHERE sID IN ( SELECT sID
               FROM Student
               WHERE GPA > 3.9 )
```

... concurrent with ...

T2

```
UPDATE Student SET GPA = (1.1)*GPA
WHERE sizeHS > 2500
```

T1; T2

or ...order matters...

T2; T1

ACID only guarantees isolation, not correctness.

Concurrent Access: Multi-Statement Inconsistency

```
INSERT INTO Archive
  SELECT * FROM Apply WHERE decision = 'N';
DELETE FROM Apply WHERE decision = 'N';
```

… concurrent with …

```
SELECT COUNT(*) FROM Apply;
SELECT COUNT(*) FROM Archive;
```

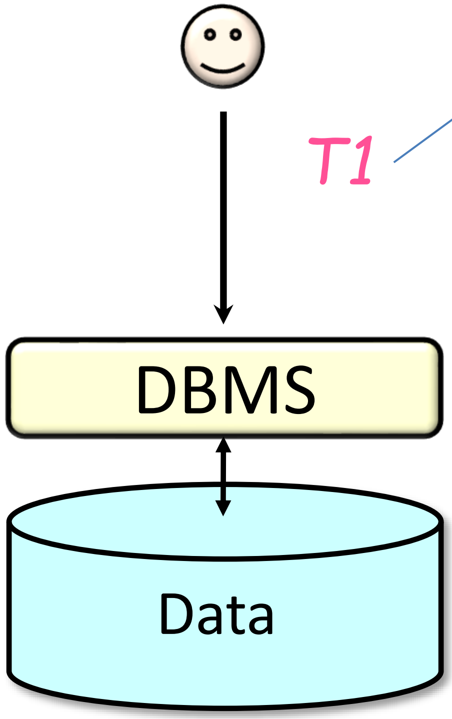
T1; T2

or *…order matters…*

T2; T1

ACID only guarantees isolation, not correctness.

ACID Properties: Durability



T1

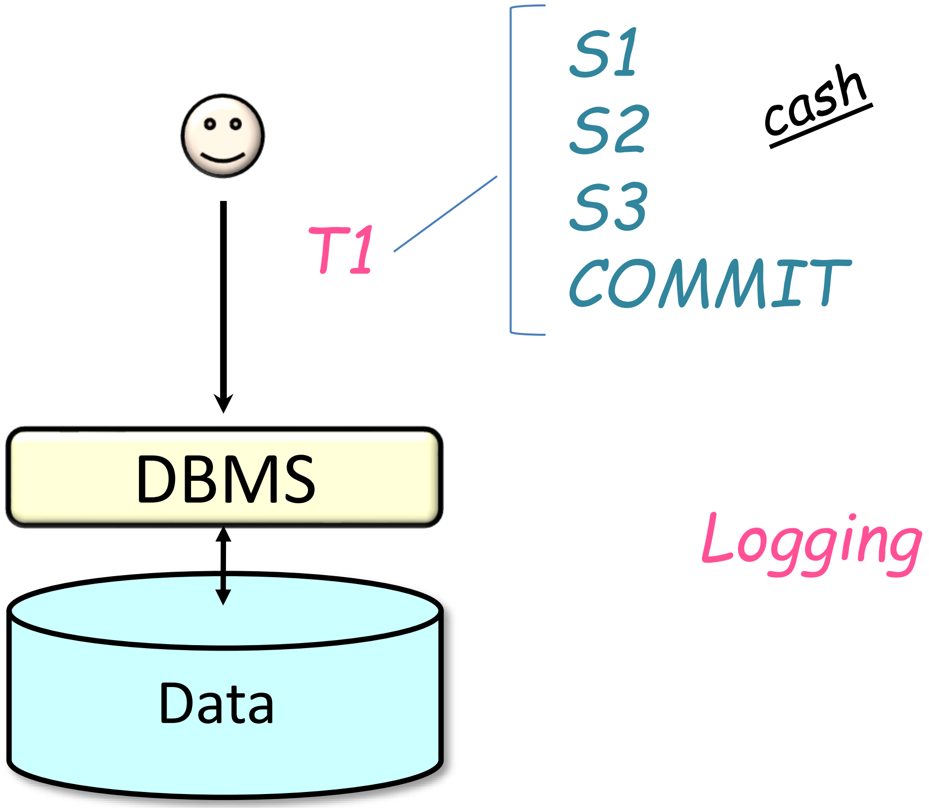
S1
S2
S3
COMMIT

cash

Logging

If system crashes after transaction commits, all effects of the transaction remain in database.

ACID Properties: Atomicity



Each transaction is 'all-or-nothing' never left half done.

Transaction Rollback (=Abort)

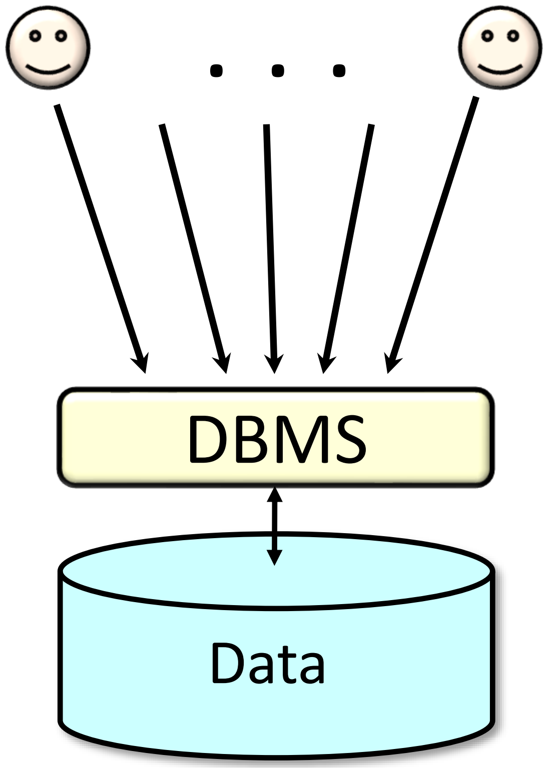
- Undoes partial effects of transaction
- Can be system- or client-initiated

Each transaction is 'all-or-nothing' never left half done.

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans = 'ok Then Commit; Else Rollback;
```

Locking

ACID Properties: Consistency



Each Client, Each Transaction:

- We can assume all constraints hold when transaction begins.
- ACID must guarantee all constraints hold when transaction ends.

Serialisability → Constrains always hold.

Solution for both Concurrency and Failures

TRANSCATIONS

Atomicity

Consistency

Isolation

Durability

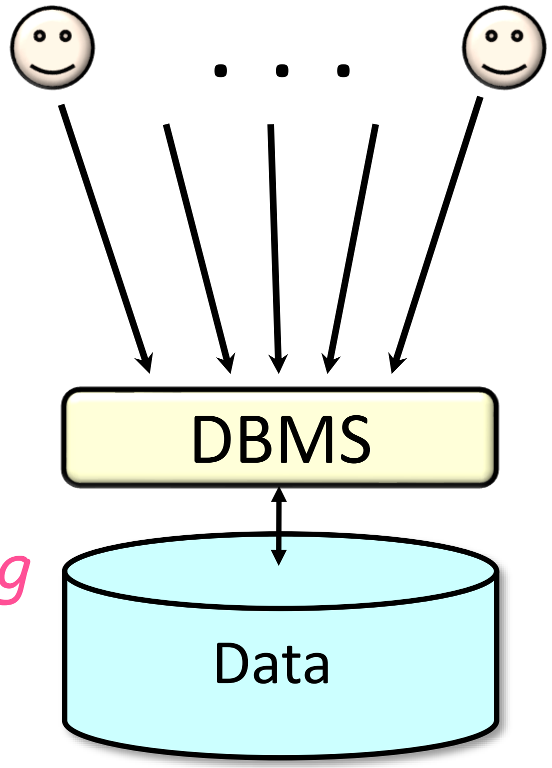
Solution for both Concurrency and Failures

TRANSCATIONS

A transaction is a sequence of one or more SQL operations treated as a unit.
SQL Standard:

- Transactions appear to run in isolation
- If the system fails, each transaction's changes are reflected either entirely or not at all.

ACID Properties: Isolation



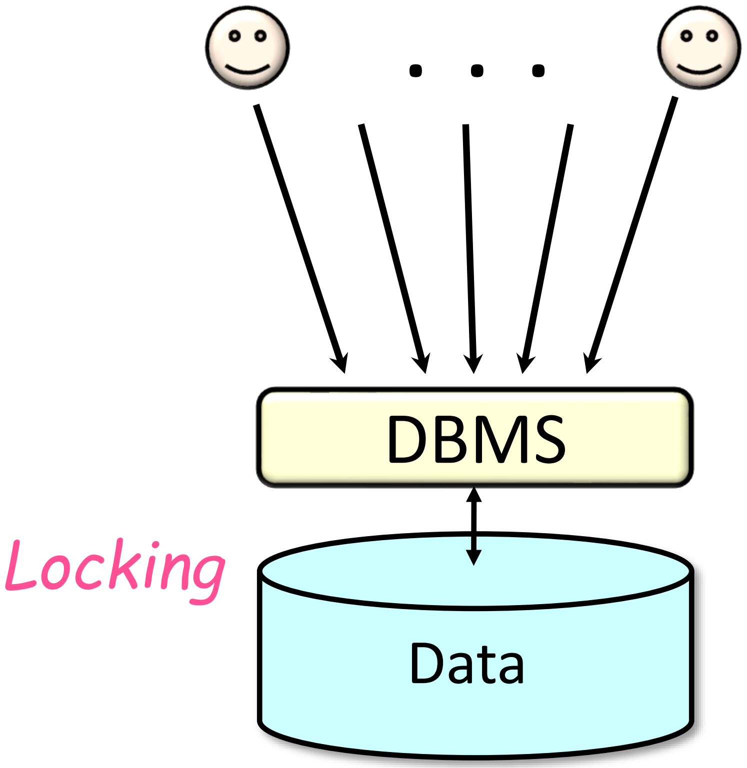
Locking

Serialisability

Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions.

→ *Overhead*
→ *Reduction in Concurrency*

ACID Properties: Isolation



Weaker 'Isolation Levels'

```
Read Uncommitted
Read Committed
Repeatable Read
```

→ *Overhead*
→ *Reduction in Concurrency*

↓ *Overhead* ↑ *Concurrency*

↓ *Consistency Guarantees*

What is the practical usage of falsifying functional dependencies in sample tables?

Constraints and Triggers

- Relational Databases
- SQL Standard

(Integrity) Constraints

- constrain allowable database states

static

Triggers

- monitor database changes
- check conditions and initiate actions

dynamic

Integrity Constraints

Impose restrictions on allowable data, beyond those imposed by structure and types

Examples

```
0.0 < GPA <= 4.0

decision : 'Y' 'N' NOT NULL

sizeHS < 200 → not admitted enrollment > 30,000
```

Integrity Constraints

Impose restrictions on allowable data, beyond those imposed by structure and types

Why use them?

- Data-Entry Errors (for Inserts)
- Correctness Criteria (for Updates)
- Enforcing Consistency
- Tell system about data – store, query processing

Integrity Constraints

Impose restrictions on allowable data, beyond those imposed by structure and types

Classification

- NOT NULL
- Key (Primary Key)
- Referential Integrity (Foreign Key)ata-Entry Errors (for Inserts)
- Attribute-Based
- Tuple-Based
- General Assertions

Declaring and Enforcing Constraints

Declaration

- with original schema
- or later

Enforcement

- check after every modification
- deferred constraint checking

Triggers

Event-Condition-Action Rules

When event occurs, check condition; if true, do action

Examples

- enrollment > 35,000 → reject all applications
- insert application with GPA > 3.95 → accept automatically

Triggers

Event-Condition-Action Rules

When event occurs, check condition; if true, do action

Why use them?

- Move logic from applications into DBMS itself
- To enforce constraints
  - Expressiveness
  - Constraint ‘Repair’ Logic

Trigger in SQL

Event-Condition-Action Rules

```
CREATE TRIGGER name
BEFORE|AFTER|INSTEAD OF events
[referencing-variables]
[FOR EACH ROW]
WHEN (condition)
action
```

Event-Condition-Action Rules

When event occurs, check condition; if true, do action

- Move monitoring logic from apps into DBMS
- Enforce constraints
  - Beyond what constraint system supports
  - Automatic constraint ‘repair’

Tricky Issues

- Row-Level vs Statement-Level
  - New/Old Row and New/Old Table
  - Before, Instead of

- Multiple triggers activated at the same time
- Trigger actions activating other triggers (chaining)
  - Also self-triggering, cycles, nested invocations
  - monitoring logic from apps into DBMS
- Conditions in WHEN vs as part of ACTION

Constraints and Triggers

- Relational Databases
- SQL Standard *systems vary considerably*

(Integrity) Constraints

- constrain allowable database states

Triggers

- monitor database changes
- check conditions and initiate actions