Atomicity: either all operations of transactions are properly reflected in DB or none are.

Consistency: consistent before & after transaction starts ie sum of total money should be SAME. inbetween

Isolation: T_j finished execution before T_i started. Multiple transaction occur independently without ^.

Durability: changes made to DB persist, even system fails.

Observable external writes: once such write has occured it cannot be undone.

Serial Schedule: (+) consistent (-) throughput will decrease.

Concurrent Schedule: (+) increase throughput (+) reduced avg response time.

① **Lost Update**

T1	T2
R(TC)	
C+=3	R(TC)
	C+=4
W(TC)	
	W(TC)

② **Tempory Read** (Dirty Read)

T1	T2
R(TC)	
C+=3	
W(TC)	
	R(TC)
	C+=4
	W(TC)
fails → R(...)	

③ **Incorrect Summary**

T1	T2
	sum=0
	avg=0
	R(C)
	sum+=C
R(A)	
A-=50	
W(A)	
	R(A)
	sum+=A
	R(B)
	sum+=B
	avg=sum/3
	commit
R(B)	
B+=50	
W(B)	
commit	

④ **Phantom Read**

T1	T2
R(X)	
	R(X)
delete(1)	
	R(1)

NON CONFLICT PAIR: R(A) R(A); R(B) R(B)

CONFLICT PAIR: R(A) W(A); W(A) R(A); W(A) W(A); R(B) W(B); W(B) R(B); W(B) W(B)

⑤ **Non Repeatable Read**

T1	T2
R(X)	
	R(X)
W(X)	
	R(X)

Shared lock (S)
S(X) Read Only

Exclusive lock (X)
X(S) R/W only

	S	X
S	✓	X
X	X	X

GRANT Request →

2PL - Two Phase
- ensure serializability
- growing & shrinking phase.

★ To check if two transactions are CONFLICT EQUIVALENT, swap non conflicting pairs

★ A schedule S is conflict serializable if it is conflict equivalent to serial schedule.

★ A schedule is conflict serializable if and only if it's precedence graph is ACYCLIC

★ Apply topological sorting on ACYCLIC to find serializable order

Irrecoverable Schedule: T_1 is read in T_2 and changes are omitted however T_1 fails, but original value cannot be recovered.

Recoverable Schedule: for each pair of T_1 and T_2 such that T_2 reads data item previously written by T_1, commit operation of T_1 appears before T_2 commit.

Cascading Rollback

T1	T2	T3
R(A)		
R(B)		
W(A)		
	R(A)	
	W(A)	
		R(A)
* fail abort		

→ all will have to ROLLBACK

★ check W-R operation before commit

Cascadeless

T1	T2
R(A)	
R(B)	
W(A)	
commit	
	R(A)
	W(A)
	commit

★ Clustering Index: if DB file contains records is sequentially ordered, search key defines order of file/PK.

★ Non clustering/Secondary: search key do not define sequential order of DB file.

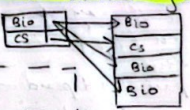
★ Dense Index: every search key value

★ Sparse Index: an index for only some SK-value. (less space/faster)

Raid

★ Dense Clustering Index

★ Dense Non Clustering:



BACKUPS

FULL

INCREMENTAL

DIFFERENTIAL

	FULL	INCREMENTAL	DIFFERENTIAL
Desc	copies entire data	full backup + changes prev backup	full backup + changes since full backup
Backup Time	Time consuming	Fast to backup	faster than full backup but slower than incremental
Recovery Time	fast recovery	slow recovery	faster than incremental but slower than full
Storage space.	requires a lot of space	less space	< full backup > incremental
Bandwidth	uses a lot bandwidth	uses less bandwidth	< full backup > incremental

MONGO DB/ NO SQL

★ Create collection
db.createCollection("cars")

★ Insertion
db.cars.insertOne()
db.cars.insertMany()

★ Read
db.cars.find() → finds all
db.cars.findOne()
db.cars.findAndModify()
db.cars.findOneAndDelete()
db.cars.findOneAndReplace()
db.cars.findOneAndUpdate()

db.collection.find({query}, {projection})

★ Update
db.cars.updateOne()
db.cars.updateMany()

★ Deletion
db.cars.deleteOne()
db.cars.deleteMany()

★ Operators
$lt → less than
$lte → less than or equal
$gt → greater than
$gte → greater than or equal
$ne → not equal
$in → in a set
$nin → not in set
$exists → matching

$or
$not
$nor
$and
$inc → increment
$eq → equal
$pop → remove 1st/last

$currentDate
$rename → renames
$set → sets value
$unset → remove field
$push → add element to array

DB LAB

★ Display all students with CS Major
db.collection.find ({'Major': 'CS'})

★ Display all students not from Khi.
db.collection.find ({'Address.City': {$ne: 'Karachi'}})

★ Display all students from Khi with CS Major.
db.collection.find ({$and: [{'Address.City': 'Karachi'}, {'Major': 'CS'}]})

★ Increment CGPA of all students by 0.1
db.students.updateMany ({}, {$inc: {'CGPA': 0.1}})

★ Display first five documents
db.collection.find({}).limit(5)

★ Display first name and age having no email
db.collection.find ({email: {$exists: false}}, {firstName: true, age: true})

★ Display only first and last name of all students.
db.collection.find ({}, {'First Name': 1, 'Last Name': 1})

★ Display all students who do not have major property
db.students.find ({'Major': {$exists: 0}})

★ Display documents from 3 to 7
db.collection.find({}).skip(2).limit(5)

★ Display all children who have "scheduled" appointments with Dr Wilson
db.collection.find ({appointments: {$elemMatch: {doctorName: "Dr Wilson", status: "Scheduled"}}})

★ Add new field weight with 0.0 value
db.collection.updateMany ({}, {$set: {"weight": 0.0}})

★ Delete all female children with email
db.collection.deleteMany ({gender: "Female", email: {$exists: 1}})

★ change field name to Contact No
db.collection.updateMany ({}, {$rename: {"phone": "contactNo"}})

NORMALISATION

→ 1NF
→ should contain all atomic attributes
→ no multivalued attributes

→ 2NF
→ find all functional dependencies of PK/candidate Key with non prime attribute and see if any of composite ones all fully functional dependent
→ if partially dependent then cannot be decomposed

→ 3NF
→ if there are any transitive dependencies.
→ we need to check two conditions for X → A.
① X is superkey
② A is prime attribute

EXAMPLE ①

• Supplier Product (SupplierID, ProductID, SupplierName, (SupplierAddress), ProductName, MonthlyRepair)

→ 1NF
Supplier Product (SupplierID, ProductID, SupplierName, SupplierCity, SupplierPostal, ProductName, MonthlyRepair)

→ 2NF
ProductID → ProductName, {SupplierID, ProductID} → MonthlyRepair
Repair (SupplierID, ProductID, MonthlyRepair)
Supplier (SupplierID, SupplierName, SupplierCity, SupplierPostal)
Product (ProductID, ProductName)

★ fully functionally dependent.

→ 3NF
Repair (SupplierID, ProductID, MonthlyRepair)
Supplier (SupplierID, SupplierName)
SupplierLocation (SupplierName, SupplierCity, SupplierPostal)
Product (ProductID, ProductName)

Topological Sorting.

→ Make precedence graph
→ Specify indegree
→ To make SERIALIZABLE
→ Pick indegree = 0
→ Erase that from precedence
→ T3 → T2 → T1

Techniques of Indexing

→ Access Type: finding record with specified attribute / in range.
→ Access time: time taken to find particular data
→ insertion time: time it takes to search the insert data
→ deletion: time taken to delete item
→ Space overhead: additional space occupied by an index structure