Natural join

- We always use $R(A, B) \bowtie S(B, C)$
- One-pass algorithms: If $B(R) \leq M$ the same techniques can be used as for other operations
- Two-pass algorithms: More problematical
- To see why, let's try to do a sort-based 2-pass algorithm for join. The first pass (sorting on B) is done as before
- For the second pass, divide (in some way — we'll discuss this later) main memory into two parts, for R and S, and assume that the tuples in the first block of each list (for R and S) *all* have the same value b for B. Then we cannot perform the join without reading some of R or S twice.

Before returning to this example, we'll look at an intermediate class of *nested-loop* or "one-and-a-half" pass algotithms.

In each algorithm, one of the arguments is read once, while the other is read more often

No restrictions on relation size

Tuple-based nested-loop join

```
FOR each tuple s in S DO
  FOR each tuple r in R DO
    IF r and s join to make a tuple t THEN
        output t ;
```

This only reads S once, but could take $T(R)T(S)$ iterations!

We'll see later a block-based variant of this algorithm that is much better

But first, an iterator version

```
Open() {
  R.Open();
  S.Open();
  s := S.GetNext();
}
GetNext() {
  REPEAT {
    r := R.GetNext();
    IF (r = NotFound) { /* R is exhausted for
                           the current s */
    R.Close()
    s := S.GetNext();
    IF (s = NotFound) RETURN NotFound
           /* both R and S are exhausted */
    R.Open();
    r := R.GetNext();
    }
  }
  UNTIL (r and s join);
  RETURN the join of r and s;
}
Close() {
  R.Close();
  S.Close();
}
```

Block-based nest-loop join

Modify the previous algorithm

- Organize access by blocks
- Use as much main memory as we can for S

We assume here that S is the smaller relation, but will not fit in main memory

$$M < B(S) \leq B(R)$$

```
FOR each chunk of M blocks of S DO BEGIN
    read these blocks into main-memory;
    FOR each block b of R DO BEGIN
      read b into main memory;
      FOR each tuple t of b DO BEGIN
        find the tuples of S in main memory that join with t;
        output the join of t with each of these tuples;
      END;
    END;
END;
```

Note that the inner loop does not involve any I/O

Example: $B(R) = 1.000$, $B(S) = 500$, $M = 100$

- Outer loop executed 5 times, each time 100 I/O
- Each iteration: Read R once, 1.000 blocks
- Total time $5 \times 1.100 = 5.500$

If we switched S and R, we iterate 10 times, 600 I/O each time, cost 6.000, so starting with the smaller relation makes sense

Analysis

- Assume S is the smaller relation
- Outer loop executed $\frac{B(S)}{M}$ times
- Each iteration reads M blocks of S and $B(R)$ blocks of R
- Total cost $B(S) + \frac{B(R)B(S)}{M}$
- If $B(S)$ is much larger than M, we can approximate this by $\frac{B(R)B(S)}{M}$
- For reasonably small relations, not much larger than a one-pass join
- For larger relations, much more expensive, but we shall see later that we can use it as a subroutine for small parts of a larger relation

Sort-based join

- First phase: Sort R and S on the join attribute B.
- Cost: $4(B(R) + B(S))$, as we are fully sorting both relations
- Using (for now) only two blocks of main memory, read the first blocks of R and S
- Find the first value of B in these blocks. if this value occurs only in one of the relations, delete all these tuples, and contine to the next value
- Read all tuples from both R and S that have this value for attribute B. Assume that they fit in the $M - 1$ remaining memory blocks
- Output all combinations of these tuples, and delete these tuples in main memory.
- We are left with only two partially filled blocks. Continue from this point

Example

- $B(R) = 1.000$, $B(S) = 500$, $M = 100$
- Sorting: $4(1.000 + 500) = 6.000$
- Provided that we never need more than 101 blocks for tuples with the same join value, the last step needs $1.000 + 500 = 1.500$ I/O
- Total: 7.500 I/O
- Nested-loop join took 5.500 which looks better
- But this is just because both relations are quite small. With $B(R) = 10.000$ and $B(S) = 5.000$, the new algorithm is much better

Hash-join

Similar to other hashing based algorithms

- Hash based on join attributes
- All tuples that join must be in the same bucket for R and the same bucket for S
- As long as these two buckets take less than M blocks (together) we can read them into main memory and join
- Complexity is similar to the previous algorithm

Index-based joins

- Idea: Use index on join attribute to find tuples that join
- Performance is usually not very good

Assume S has an index on B

- Examine each block of R
- Consider a tuple t in this block
- Let t_B be the value of the B-attribute
- Use the index to find all the tuples in S with this value for B
- Output the join of these tuples with r

Analysis

- Accessing R: $B(R)$ I/O
- For each tuple of R, on average $\frac{T(S)}{V(S,B)}$ have the same value for attribute B
- This must be done for each tuple of R, yielding cost $\frac{T(R)T(S)}{V(S,B)}$
- Total $B(R) + \frac{T(R)T(S)}{V(S,B)}$

Example

The use of $T(R)$ and $T(S)$ already should show that this is very expensive

- $B(R) = 1.000$, $B(S) = 500$
- 10 tuples per block, so $T(R) = 10000$ and $T(S) = 5.000$
- Assume $V(S, B) = 100$
- Cost is $1000 + \frac{10.000 \times 500}{100} = 50.000$
- Much higher than previous algorithms!
- Might be useful if R is much smaller than S

How can we improve this

Use a sorted index, in which we get the tuples in order

Example

- Suppose that we have sorted indexes for B on both R and S
- Let the B-values for the tuples in R be 1, 3, 4, 4, 5, 6
- Let the B-values for the tuples in S be 2, 2, 4, 4, 6, 7
- Find the first tuple in both relations
- Since $1 < 2$ go on to the next tuple in R
- Since $3 > 2$, go on to the next tuple in S
- Since $3 < 4$ go on to the next tuple in R
- Retrieve tuples in R and S and join them

Note that all except the last part only use the index

Assuming that all tuples that match in the last step fit in memory, this costs basically $T(R) + T(S)$

With a clustered index, we can read a block at a time, with cost $B(R) + B(S)$ (plus a small cost for reading the index)

Physical query plan

We outline remaining steps, namely

- Decisions whether to materialize intermediate results (store on disk) and when to pipeline to the next operator
- A notation for physical-query-plan operators, including access methods for relations, and algorithms for relational-algebra operators

Pipelining vs. materialization

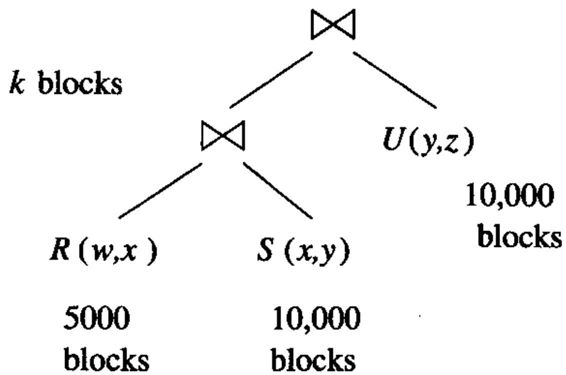
- Naive approach: implement each operation and store the result on disk until it is needed
- It is usually better to interleave the execution of several operations. The tuples produced by one operation are passed directly to the operation that uses it, without storing tuples on disk
- Piplining is typically implemented by a network of iterators which call each other when needed
- This saves a lot of disk I/O
- Disadvantage: Less memory available for each operation

Pipeling unary operations

- Projection and selection
- One block in main memory used for a block of data
- The next operator is implemented in such a way that `GetNext()` gets the first tuple in main memory, instead of accessing the disk

Pipelining binary operations

- Use one block to pass result to next operator
- Number of blocks to compute and to use the result can vary, depending on size of result and sizes of arguments
- Example
$$(R(w, x) \bowtie S(x, y)) \bowtie U(y, z)$$
- $B(R) = 5.000$, $B(S) = B(U) = 10.000$
- $B(R \bowtie S) = k$.
- The algorithm depends on the value of k
- Joins implemented as hash-joins
- $M = 100$



- $R \bowtie S$. Neither relation fits in memory, so we use 2-pass hash-join
- First pass: Partition R into 100 buckets
- Each bucket takes 50 blocks
- Second pass of $R \bowtie S$ uses 51 blocks:
  - 50 blocks for a bucket of R
  - One block of S
- 50 left to use for the join with U

First case: $k \leq 49$

- Assume $k \leq 49$
- Pipeline them into 49 buffers, organize as hash table
- One block left for U
- So second join can be evaluated in one pass
- Cost
  - 45.000 for the 2-pass hash-join of R and S
  - 10.000 to read U
- Total cost 55.000

Second case: $49 < k < 5.000$

- Why 5.000?
- $5.000 = 100 \times 50$, i.e., $M \times 50$
- This means that if we use a 50-bucket hash-join algorithm for joining $(R \bowtie S)$ and U, we can fit an entire bucket of $R \bowtie S$ in main memory
- The first step is to hash U into 50 buckets (each one takes 200 blocks)
- Now join R with S via two pass hash-join as before
- This uses 51 blocks, leaving 50 free. These blocks will be used to store the 50 buckets of $R \bowtie S$
- Whenever we generate a tuple in $R \bowtie S$, put in in the appropriate main-memory block, writing blocks to disk when they are full

- Now join $R \bowtie S$ with U bucket by bucket
  - Buckets of $R \bowtie S$ take 100 blocks
  - Load one block of the corresponding bucket of U at each step
  - Total memory needed at each step: 101 blocks
- Costs
  - Read U and write tuples into buckets: 20.000
  - Two-pass hash-join of R and S: 45.000
  - Write out buckets of $R \bowtie S$: k
  - Reading both relations in final join: k+10.000
- Total $75.000 + 2k$
- Big jump from 49 to 50 (but we could probably still use 1-pass algorithm with some thrashing)

Third case: $k > 5.000$

- Note that we can't use a 2-pass algorithm in 50 buffers
- We could use a 3-pass one
- We haven't studied these. The cost would be $20.000 + 2k$ I/O, but we'll see how to do better *without* pipelining
- Comute $R \bowtie S$ using 2-pass hash-join and store on disk
- Cost $45.000 + k$
- Join $R \bowtie S$ with U using 2-pass hash-join with U
- Since $B(U) = 10.000$, using 100 buckets we can fit a bucket in memory (we are using U as the *left* argument)
- This join then costs $30.000 + 3k$ (read each argument twice, write once)
- Total cost $75.000 + 4k$

Summary

Range	Pipeline?	Final join	I/O
$k \leq 49$	Pipeline	1-pass	55.000
$50 \leq k \leq 5.000$	Pipeline	50-bucket 2-pass	$75.000 + k$
$5.000 < k$	Materialize	100-bucket 2-pass	$75.000 + 4k$

Physical Query Plans: Notation

- Each operator becomes an operator of the physical plan
- Each leaf becomes a scan operator applied to a relation
- Materialized relation indicated by `Store` operator, plus the appropriate scan operator (`TableScan` unless we have constructed an index)
- We skip the latter detail, and indicate materialization by double lines ("crossing out"); all others are pipelined

Operations for leaves

Possible operations on leaves (base relations)

- `TableScan(R)`: Read all blocks in arbitrary order
- `SortScan(R,L)`: Tuples read in sorted order, using the attributes in L
- `IndexScan(R,C)`: C is a condition of the form $A < c$, $A = c$ or $A > c$. Tuples are accessed using an index on A
- `IndexScan(R, A)`: The entire relation R is retrieved via an index on $R.A$ (identical to `TableScan`, as far as we are concerned)

Selection

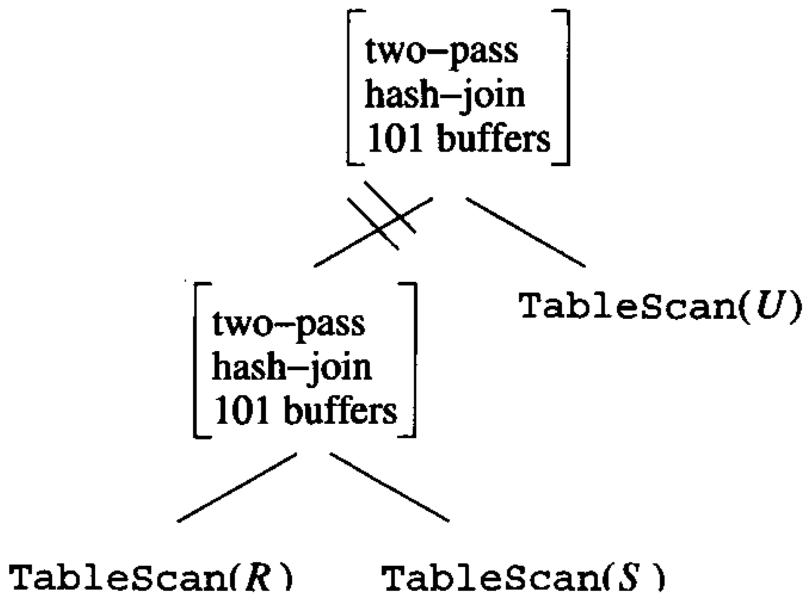
- $\sigma_C(R)$ can be combined with the access method
- If the argument is not a stored relation, or we have no index, we use the `Filter` operator
- Conjunction, e.g. $A\theta c$ `AND` D with index on `R.A`:
  - Use `IndexScan(R,C)` to access R with first condition followed by
  - `Filter(D)`

Physical Sort Operators

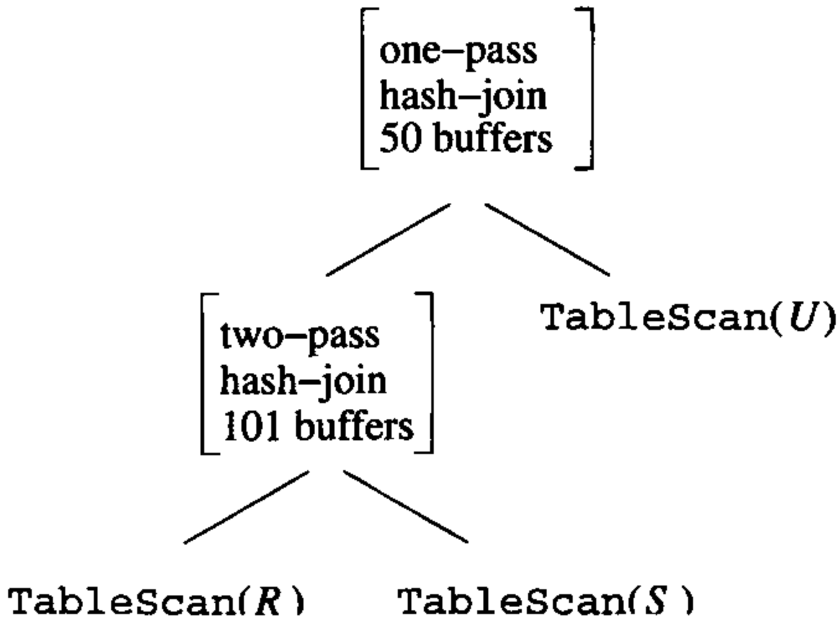
- `SortScan(R,L)`
- Sort-based join or grouping: use an explicit `Sort(L)`
- This can also be used for `ORDER BY` clause

Other operations. Specify

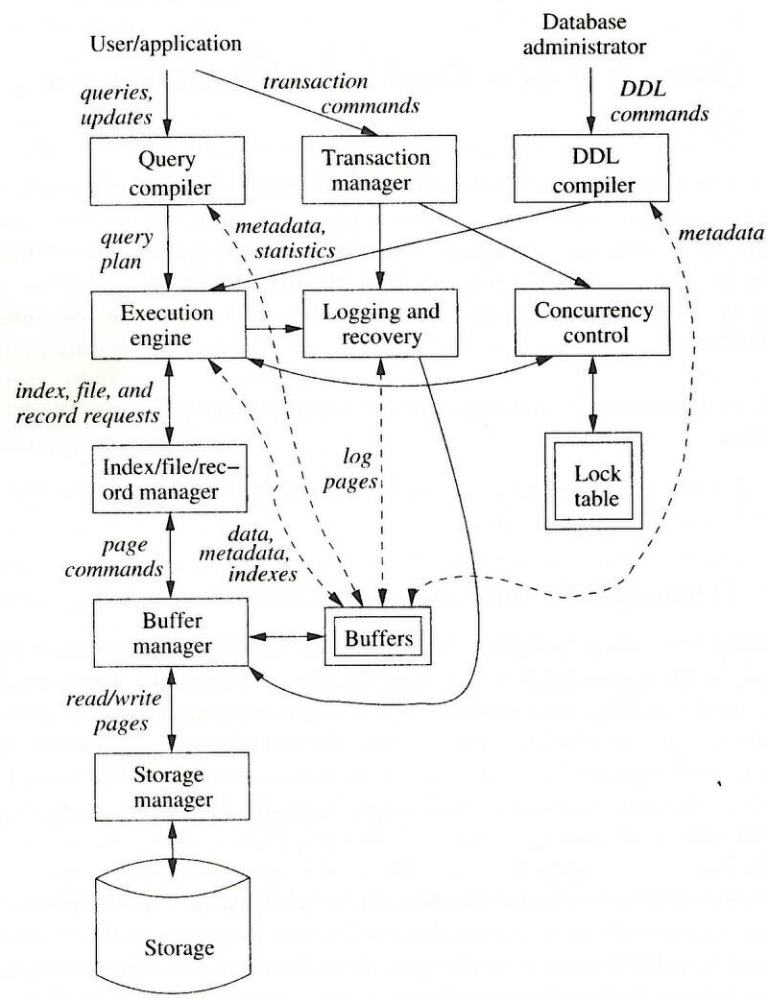
- The operation being performed (join, grouping, etc.)
- Parameters (condition in theta-join etc.)
- Strategy (sort-based etc.)
- Number of passes (can be left until run-time)
- Expected number of memory blocks

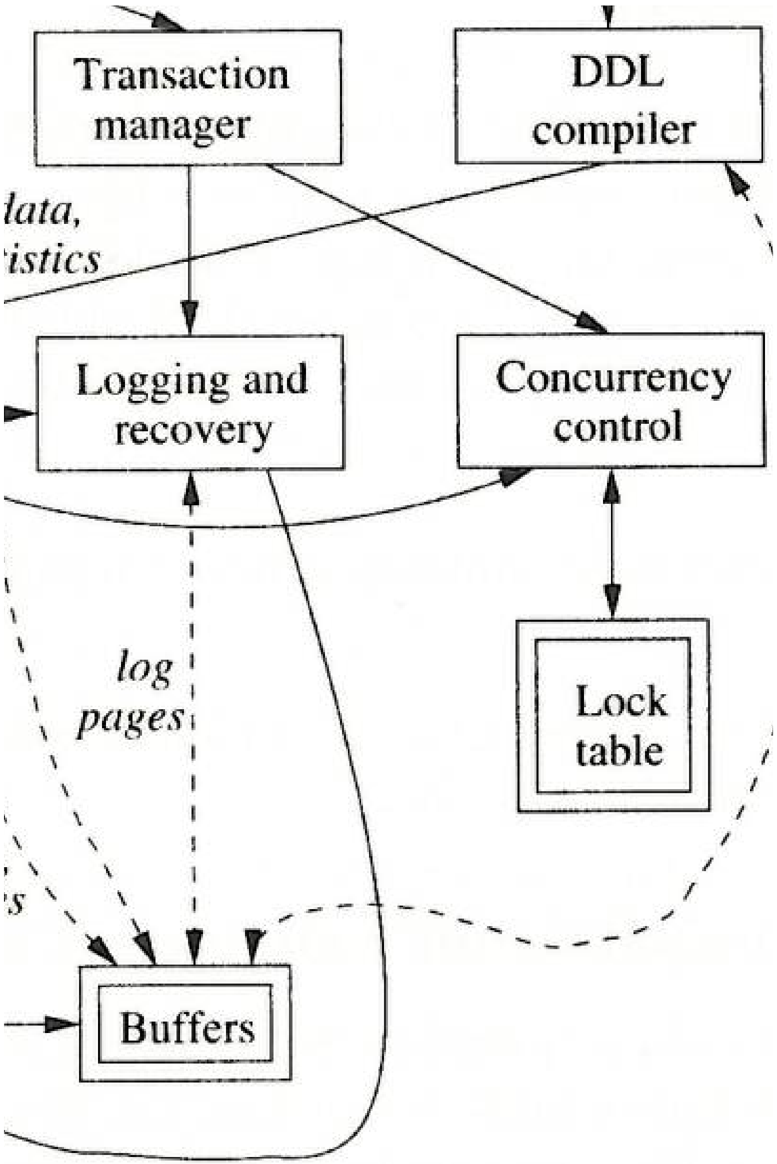


- Figure shows our example when $k > 5.000$
- Access each of the relations by a table-scan
- Use two-pass hash-join for the first join and materialize it (crossed lines)
- Use two-pass hash-join for the second join (materialization implies table-scan)



- The case $k < 49$
- Second join has a different number of passes, a different number of memory blocks
- Left argument is pipelined


User/application
Database administrator
queries, updates
transaction commands
DDL commands
Query compiler
Transaction manager
DDL compiler
query plan
metadata, statistics
metadata
Execution engine
Logging and recovery
Concurrency control
index, file, and record requests
Index/file/rec–ord manager
log pages
Lock table
page commands
data, metadata, indexes
Buffer manager
Buffers
read/write pages
Storage manager
Storage



Transaction manager
DDL compiler
lata,
istics
Logging and recovery
Concurrency control
log pages
Lock table
s
Buffers

Concurrency control

- Our model so far: one user queries or modifies the database
- Operations on the database are executed one at a time, and the database state left after an operation is the state upon which the next acts
- Operations are "atomic": It is impossible for the system to fail in the middle of a modification, leaving the database in an inconsistent state
- Real life is different
- In applications like Web services, banking, or airline reservations, hundreds of operations per second may be performed on the database
- Two operations affecting the same bank account or flight may be executed at the same time, and may might interact in strange ways

Example

- Example of what could go wrong
- Note that we combine the DBMS with the O/S to get an intuitive example (the DBMS really only deals with SQL statements, but similar problems arise even there)
- Airline website where customers chose a seat
- Relation

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

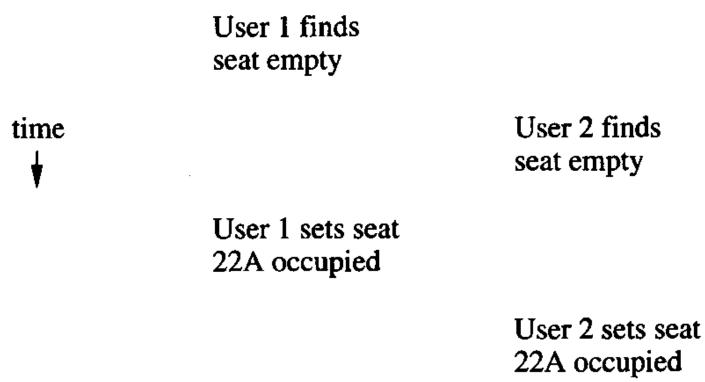
- Query

```
SELECT seatNo
FROM Flights
WHERE fltNo = 123 AND fltDate = DATE 2008-12-25
      AND seatStatus = available;
```

- Customer clicks on empty seat, say 22A
- System reserves this seat, and modifies the database

```
UPDATE Flights
SET seatStatus = occupied
WHERE fltNo = 123 AND fltDate = DATE 2008-12-25
     AND seatNo = 22A;
```

- Suppose another customer is using the system and the same time
- He also sees seat 22A empty and clicks on it



- Both customers believe they have got seat 22A
- Only the second one “really” has it
- In SQL: *transaction*, a group of operations that must be performed together
- If these are defined as transactions, the two transactions must behave as though they were run *serially*
- But they don’t have to be actually run serially; we can allow overlap if it doesn’t create problems
- The more overlap, the better the performance of the system
- A sequence of operations (as in our example) is *serializable* if it is equal to a serial execution of the transactions

Another example

- Bank account records

  `Accounts(acctNo, balance)`

- Transferring \$100 from account 123 to account 456
- First check whether there is at least \$100 in account 123
- If so, we execute the following two steps:
  - Add \$100 to account 456:

    ```
    UPDATE Accounts
    SET balance = balance + 100
    WHERE acctNo = 456;
    ```

  - Subtract \$100 from account 123:

    ```
    UPDATE Accounts
    SET balance = balance - 100
    WHERE acctNo = 123;
    ```

Possible problem

- Suppose there is a failure between the two steps
- The database is left in a state where money has been transferred into the second account, but taken out of the first account
- The two updates should be done atomically: either both are done or neither

Transactions

- Solution: group database operations into *transactions*
- A transaction is a collection of one or more operations on the database that must be executed atomically: either all operations are performed or none are
- SQL (default) requires that transactions are executed in a serializable manner: equivalent to one transaction executed in its entirety before the other
- Note that this does not specify *which* transaction is executed first
- The DBMS may allow the user to specify less stringent constraints on the interleaving of operations from two or more transactions

Outline

We study

- Theory of serializability: how to test
- Specification in SQL
- Theory of locking: how to avoid nonserializable behaviour

Testing serializability

- Schedule: An execution of transactions with operations interleaved

T_1	T_2
$\texttt{READ}(A, t)$	$\texttt{READ}(A, s)$
$t \leftarrow t + 100$	$s \leftarrow s * 2$
$\texttt{WRITE}(A, t)$	$\texttt{WRITE}(A, s)$
$\texttt{READ}(B, t)$	$\texttt{READ}(B, s)$
$t \leftarrow t + 100$	$s \leftarrow s * 2$
$\texttt{READ}(B, t)$	$\texttt{READ}(B, s)$

- Column 1 is transaction T_1, column 2 T_2
- Nothing is implied about execution time
- Invariant: If $A = B$ before executing a transaction, it also holds afterwards
- The invariant is not part of the theory: it is only used as an easy way to see when an execution is wrong

Serial schedules

- Our tables now describe which operation is executed at each step, from top to bottom
- We also list the values of A and B at each step
- There are two serial executions: T_1 followed by T_2 and T_2 followed by T_1

Serial schedule

T_1	T_2	A	B
		25	25
$\texttt{READ}(A, t)$			
$t \leftarrow t + 100$			
$\texttt{WRITE}(A, t)$		125	
$\texttt{READ}(B, t)$			
$t \leftarrow t + 100$			
$\texttt{READ}(B, t)$			125
	$\texttt{READ}(A, s)$		
	$s \leftarrow s * 2$		
	$\texttt{WRITE}(A, s)$	250	
	$\texttt{READ}(B, s)$		
	$s \leftarrow s * 2$		
	$\texttt{READ}(B, s)$		250

Another serial schedule

Note that the final results are different

T_1	T_2	A	B
		25	25
	$\texttt{READ}(A, s)$		
	$s \leftarrow s * 2$		
	$\texttt{WRITE}(A, s)$	50	
	$\texttt{READ}(B, s)$		
	$s \leftarrow s * 2$		
	$\texttt{READ}(B, s)$		50
$\texttt{READ}(A, t)$			
$t \leftarrow t + 100$			
$\texttt{WRITE}(A, t)$		150	
$\texttt{READ}(B, t)$			
$t \leftarrow t + 100$			
$\texttt{READ}(B, t)$			150

Serialiazble schedule

T_1	T_2	A	B
		25	25
$\texttt{READ}(A, t)$			
$t \leftarrow t + 100$			
$\texttt{WRITE}(A, t)$		125	
	$\texttt{READ}(A, s)$		
	$s \leftarrow s * 2$		
	$\texttt{WRITE}(A, s)$	250	
$\texttt{READ}(B, t)$			
$t \leftarrow t + 100$			
$\texttt{READ}(B, t)$			125
	$\texttt{READ}(B, s)$		
	$s \leftarrow s * 2$		
	$\texttt{READ}(B, s)$		250

- $A = B$ holds
- This doesn't prove serializability, but we shall see later that it is indeed equivalent to (T_1, T_2)

Non-serializable schedule

T_1	T_2	A	B
		25	25
$\texttt{READ}(A, t)$			
$t \leftarrow t + 100$			
$\texttt{WRITE}(A, t)$		125	
	$\texttt{READ}(A, s)$		
	$s \leftarrow s * 2$		
	$\texttt{WRITE}(A, s)$	250	
	$\texttt{READ}(B, s)$		
	$s \leftarrow s * 2$		
	$\texttt{READ}(B, s)$		50
$\texttt{READ}(B, t)$			
$t \leftarrow t + 100$			
$\texttt{READ}(B, t)$			150

$A \neq B$!

A slightly different example

T_1	T_2	A	B
		25	25
READ(A, t)			
$t \leftarrow t + 100$			
WRITE(A, t)		125	
	READ(A, s)		
	$s \leftarrow s + 200$		
	WRITE(A, s)	325	
	READ(B, s)		
	$s \leftarrow s + 200$		
	READ(B, s)		225
READ(B, t)			
$t \leftarrow t + 100$			
READ(B, t)			325

It can be shown that this is equivalent to (T_1, T_2)

Testing

- Analysing the details of the arithmetic is very difficult
- It is quite rare to have a situation like the previous one
- We therefore ignore the details, and test if the schedule will be serializable, no matter what the program does
- We analyze *only* the sequence of reads and write
- This means we miss schedules like the previous one

General principle

- Our algorithms must not give *false positives*: Saying a schedule is serializable when it is not
- A few *false negatives* are OK. Saying a schedule is not serializable when it is simply reduces performance, but never gives false behaviour
- So we seek to design algorithms with no false positives, and with a few false negatives as we can
- Analyzing only the sequences of reads and writes is our first example of this rule

Notation

- $r_T(X)$: Transaction T reads database element X
- $w_T(X)$: Transaction T writes database element X
- If the transactions are T_1, T_2, $\ldots$, we write $w_i(X)$ instead of $w_{T_i}(X)$
- Example of transactions.
- T_1:

$$r_1(A); w_1(A); r_1(B); w_1(B)$$

- T_2:

$$r_2(A); w_2(A); r_2(B); w_2(B)$$

- Schedule (serializable):

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$

Notation

- An action is an expression of the form $r_i(X)$ or $w_i(X)$
- A transaction T_i is a sequence of actions with subscript i
- A schedule S of a set of transactions T is a sequence of actions, in which, for each transaction T_i in T , the actions of T_i appear in S in the same order that they appear in the definition of T_i itself
- We say that S is an interleaving of the actions of the transactions of which it is composed.

Conflict-serializability

Given a schedule S, examine *consecutive* actions from different transactions T_i and T_j $(i \neq j)$

- $r_i(X); r_j(Y)$ is never a conflict, since neither changes the value of any database element
- $r_i(X); w_j(Y)$, $X \neq Y$ is not a conflict: T_j writes Y before T_i reads X, the value of X is not changed. Furthermore the read of X by T_i has no effect on T_j
- $w_i(X); r_j(Y)$, $X \neq Y$ is not a conflict for the same reason
- $w_i(X); w_j(Y)$, $X \neq Y$ is not a conflict for similar reasons

If two actions do not conflict, we can swap them without changing the results

Conflicts

- Two actions of the same transaction, such as $r_i(X); w_i(Y)$ always conflict. We cannot change the order of actions in a single transaction
- $w_i(X); w_j(X)$ is a conflict. The value of X is whatever T_j computed. If we swap the order, we leave X with the value computed by T_i
- $r_i(X); w_j(X)$ is a conflict. If we swap the order, the value of X read by T_i will be that written by T_j, which may be different from the previous value of X
- $w_i(X); r_j(X)$ is a conflict, for similar reasons

- Any two actions of different transactions may be swapped unless:
    - They involve the same database element, and
    - At least one is a write.
- Given schedule, we make as many nonconflicting swaps as needed, with the goal of turning the schedule into a serial schedule.
- If we can do so, then the original schedule is serializable
- Two schedules are *conflict-equivalent* if each can be turned into the other by a sequence of nonconflicting swaps of adjacent actions
- A schedule is *conflict-serializable* if it is conflict-equivalent to a serial schedule
- Conflict-serializability implies serializability
- The converse is false, but conflict-serializability is usually sufficient in pratice (remember that a few false negatives are not a problem)

Converse is false

T_1, T_2, and T_3

- T_1: $w_1(Y); w_1(X)$
- T_2: $w_2(Y); w_2(X)$
- T_3: $w_1(X)$
- S_1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X)$ is serial
- S_2: $w_1(Y); w_2(Y); w_2(X); w_1(S); w_3(X)$ is not serial
- S_2 is not conflict-serializable
- But the two are equivalent, as the final effect is for X to get the value written by T_3; T_1 and T_2 are completely irrelevant

Example

S: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

S is conflict serializable, and equivalent to (T_1, T_2)

Proof

$$
\begin{array}{llllllll}
r_1(A); & w_1(A); & r_2(A); & \underline{w_2(A)}; & \underline{r_1(B)}; & w_1(B); & r_2(B); & w_2(B) \\
r_1(A); & w_1(A); & \underline{r_2(A)}; & \underline{r_1(B)}; & w_2(A); & w_1(B); & r_2(B); & w_2(B) \\
r_1(A); & w_1(A); & r_1(B); & \underline{r_2(A)}; & \underline{w_1(B)}; & w_2(A); & r_2(B); & w_2(B) \\
r_1(A); & w_1(A); & r_1(B); & w_1(B); & r_2(A); & w_2(A); & r_2(B); & w_2(B)
\end{array}
$$

Testing conflict-serializability

Given schedule S, involving transactions T_1 and T_2 (and maybe others) T_1 *takes precedence over* T_2, written $T_1 <_S T_2$, if there are actions A_1 of T_1 and A_2 of T_2 such that

- A_1 is ahead of A_2 in S
- Both A_1 and A_2 involve the same database element
- At least one of A_1 and A_2 is a write action.

Note that these are exactly the conditions under which we cannot swap the order of A_1 and A_2

Therefore A_1 appears before A_2 in any schedule that is conflict-equivalent to S

So any conflict-equivalent serial schedule must have T_1 before T_2

Precedence graph

A directed graph

- Nodes: transactions of S
- Edge from node T_i (written i) to T_j when $T_i <_S T_j$

Example:

S: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

- $T_2 <_S T_3$ due to $r_2(A)$ being ahead of $w_3(A)$ (and other reasons)
- $T_1 <_S T_2$ due to $r_1(B)$ being ahead of $w_2(B)$

Graph

Algorithm

- Construct the precedence graph
- Test if it has (directed) cycles
- If it does, S is not conflict serializable
- If not, any topological order of the nodes is a conflict-equivalent serial order

- Our previous graph is acyclic, so the schedule is serializable
- Ony one order possible: $(T_1, T_2.T_3)$
- We can convert the schedule:

  S: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$
- Into

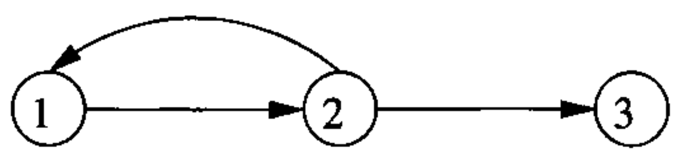
  S': $r_1(B); w_1(B); r_2(A); r_2(B); w_2(B); w_2(A); r_3(A); w_3(A)$
- By a series of local swaps

Another example

- Consider the schedule

  S: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

- A: $T_2 \prec_S T_3$
- B: $T_1 \prec_S T_2$, $T_2 \prec_S T_1$
- Graph is cyclic, so schedule is not conflict-serializable



Why this works

First direction: If there is a cycle, the schedule is not serializable

- Suppose there is a cycle of length n, $T_1 \to T_2 \to \cdots \to T_n \to T_1$
- Then the actions of T_1 must precede those of T_2, which must precede those of T_3 etc.
- In this way, the actions of T_1 must precede those of T_1, a contradiction

Converse: Proof by induction on the number of actions

- Induction: If the graph for n has no cycles, we can reorder the schedules actions using legal swaps so the schedule becomes serial
- Basis: $n = 1$. Schedule must allready be serial
- Induction. Assume the claim true for $n = 1$. We prove it for n
- Schedule consists of actions of transactions $T_1, \ldots, T_n$ with acyclic graph S
- If the graph is acyclic, it must have a node i with no incoming arcs
- No incoming arcs means that no node precedes i in our ordering

- But this means that there is no action involving another transaction T_j that
  - Precedes some action of T_i and
  - conflicts with that action
- We can then move all the actions of T_1 to the beginning of the schedule, getting

  (Actions of T_i)(Actions of the remaining $n-1$ transactions)

- Consider the second part
- The relative order of actions is preserved, so the graph is the original graph with i deleted
- This graph must be acyclic, so we can apply the inductive hypothisis
- This yields a serial orderimg