Database and Information Systems

Michal Krátký & Radim Bača

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2020/2021

Content

Recovery management

Recovery means a database recovery from an unexpected failure (wrong attribute value, hardware failure, etc.).

- **Recovery** is not a part of every database management system.
- The result of recovery is a database in a **correct state**.
- We use some **redundant information** (hidden for a user) to ensure the correct state.
  - RDBMS utilizes redundant information during the database recovery to get a database from an incorrect state in the correct state.

Transactions

Transaction is a sequence of operations (inseparable, atomic) which starts with **BEGIN TRANSACTION** and ends with the **COMMIT** or **ROLLBACK** operation. Therefore, all operations from the sequence are performed or none.

Example of a transaction

We want to transfer 100€ from one account to another account.

```
BEGIN TRANSACTION;

try {
  UPDATE Account 345 { balance -= 100; }
  UPDATE Account 789 { balance += 100; }
  COMMIT;
}
catch(SqlException) {
  ROLLBACK;
}
```

Clearly, we have to proceed both or none of the operations.
After processing of the first operation the database is in an incorrect state: 100€ is loosed.

Properties of transaction

- Usually, a transaction involves **more than one** operation.
- The goal of a transaction is to transform a database from one correct state into another correct state.
- The database does not have to be in a correct state during the transaction processing.

Notice: to satisfy the correctness, we have to process both operations `UPDATE` from the previous example.

Recovery

- What kind of failures can occur during the transaction processing?
- Failures:
  - **Local failures**: a bug in an SQL command
  - **Global failures**
    - System failures (soft crash): electricity failure, crash of operating system
    - Hardware failures (hard crash): HDD failure
- **Transaction Manager** (or transaction processing monitor) takes care about the transaction management.

Transaction commands

A transaction is handled using the following commands:

- **COMMIT** – signalizes a successful end of the transaction. Programmer sends a signal to a transaction manager that the transaction was successfully finished and the database in a correct state. All changes made during the transaction can be safely stored in a database. In other words, all changes are committed in the database.
- **ROLLBACK** – signalizes a failure during the transaction processing. Programmer sends a signal to a transaction manager that all changes made during the transaction has to be cancelled (undone).

Transaction, example


```
BEGIN TRANSACTION;

try {
  UPDATE Account 345 { balance -= 100; }
  UPDATE Account 789 { balance += 100; }
  COMMIT;
}
catch(SqlException) {
  ROLLBACK;
}
```

How to cancel updates?

- We **do not directly work** with the data file.
- RDBMS has several components and the SQL processor does not make changes directly in the data file on a secondary storage (hard disk).
- DBMS utilizes **a log file** to support the COMMIT and ROLLBACK operations, which capture all changes made during the transaction.
  - DBMS can cancel the changes using the log file when the ROLLBACK operation is invoked.

Transaction, properties

- Recovery and transaction management cause time overhead, however, a lot of applications can not work reliably without it.
- Recovery is very closely related to concurrency control. We do not consider it now.
- Transaction is an atomic operation – a transaction can not be processed within another transaction.

Correct vs. consistent database?

- **Consistent** means that there are no exceptions from integrity constraints.
- For example, we are not able to define integrity constraints in the bank account example.
- We say that a transaction is a sequence of operations which transforms a database from one correct state to another one.
- Simply, **the correct state** reflects the result of operations processed in the real world.

Commit point 1/2

- A transaction starts with **BEGIN TRANSACTION** and ends with **COMMIT** or **ROLLBACK**.
- **Commit point** (the time point of the commit) corresponds to a successful end of a transaction and it marks a correct state of the database.
- Therefore, ROLLBACK returns the database to the last commit point (the last correct state).

Commit point 2/2

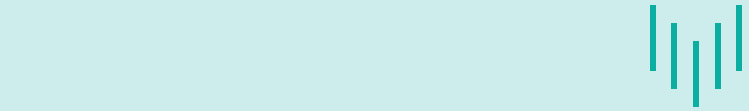
- **The record of a commit point** is written into a log file when the commit point is reached.
- All changes made by the transaction are successfully stored in a database.
- All directories and locks are released.

ACID

Every transaction has to support **ACID** properties:

- **Atomicity** – the transaction has to be atomic: all operations are performed or none.
- **Correctness** – the transaction transforms the database from one correct state into another one. During the transaction processing, database can be temporarily in an incorrect state.
- **Isolation** – two transactions running concurrently can not see the state of each other (data in an intermediate state) until they are committed.
- **Durability** – changes made by a committed transaction are stored in the database and they can not be undone.

Implementation of DBMS



1. **All data are stored in the main memory** $\Rightarrow$ the system is fast however after a system failure all updates are lost.

2. **All data are stored on disk** $\Rightarrow$ the system is slow.

   Throughput:

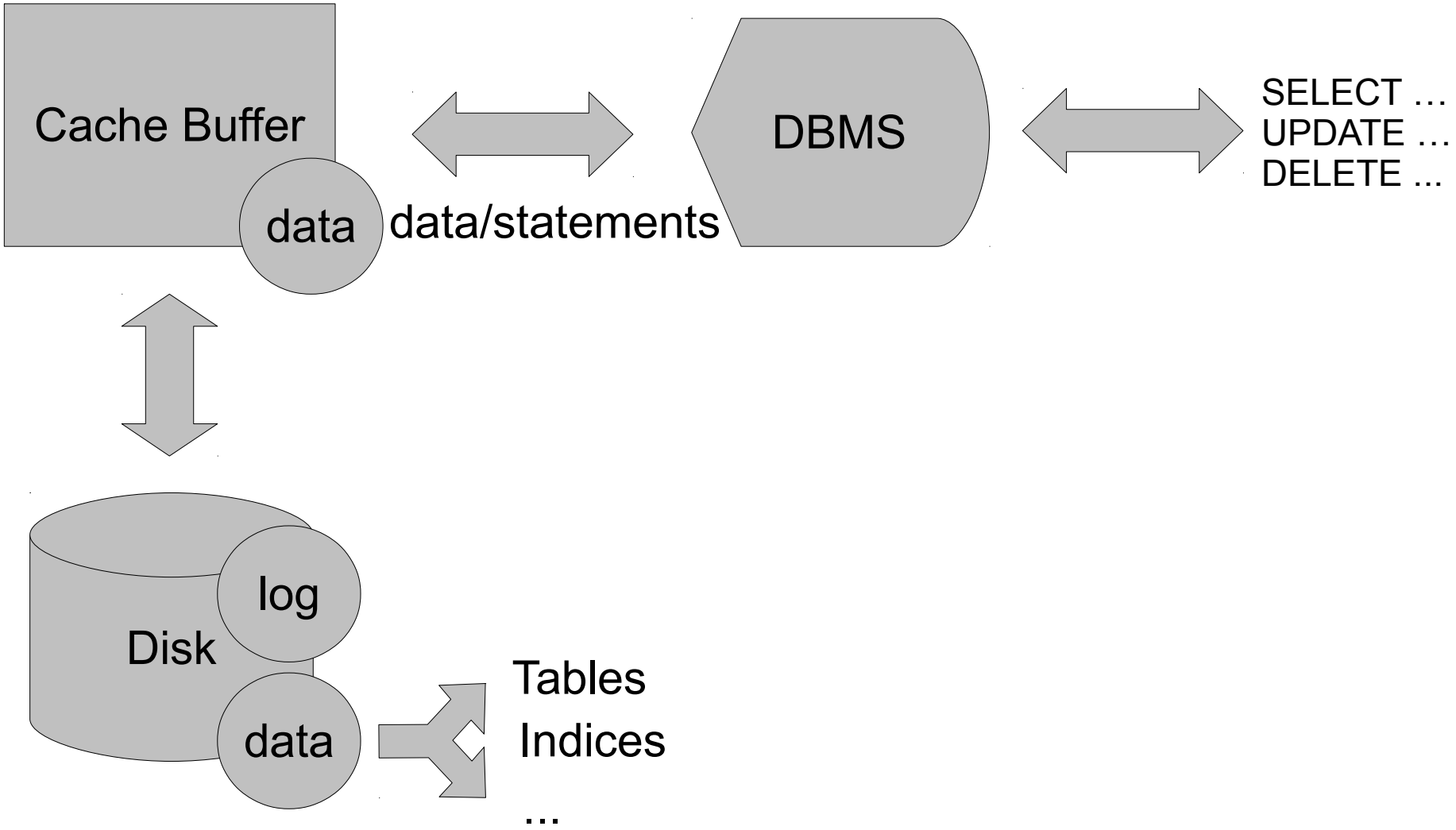
   - The main memory: up-to GB/s
   - Sequential reading/writing on disk: up-to 2 GB/s, but standard maximal value is 600 MB/s.
   - Random reading/writing: hundreds kB – MB/s (SSD provides a higher throughput but it still means a problem).

A real implementation utilizes the main memory as often as possible, however log file and database are stored on a disk.

Recovery implementation

- Due to an efficiency reasons, RDBMS uses a **cache buffer** in the main memory containing actual records.
- It can happen that the committed records were not written **into the database on a disk**, however, the system has to be recoverable.
- **Write-ahead log rule (WAL)**: All changes has to be written into the log file before data are written into the database (the commit record is written after all changes are written).
- **Result:** DBMS is able to recover the database using the log file.

DBMS Architecture



Log vs data

- Why **it is not possible to update a database** (stored on disk) but we can update the log file (stored on disk) from the efficiency point of view?
- Updates of the database often mean the use of random reads/writes to change some data structures (the throughput can be hundreds kB/s).
- The log file is updated by sequential writes (the throughput can be hundreds MB/s, in the case of a special SSD, up/to 1GB/s).

System recovery

- We recover the whole system not only one transaction.
- The system recovery is necessary after a global failure.
- The system recovery influences **all transactions** running during a global failure.
- A part of the database can be destroyed in the case of a secondary storage failure.

System recovery

- A basic problem of a system failure is a lost of the RDBMS's buffer content:
  - The state of an unfinished transaction is unknown and we have to cancel the transaction in the log file. We call it the **UNDO** operation.
  - In some cases a transaction is committed, however, the updates are stored only in the log file, not in the database. Therefore it is necessary to store the updates in the database. We have to call the **REDO** operation.

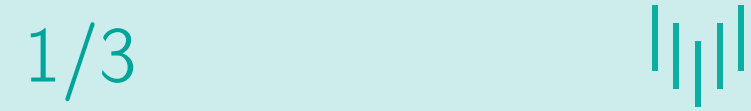
Basic recovery techniques

We described basic concepts of recovery, now we introduce three basic recovery techniques:

1. Recovery using a **deferred update** (**NO-UNDO/REDO**),
2. Recovery using an **immediate update** (**UNDO/NO-REDO**).
3. A combination of both techniques (**UNDO/REDO**).

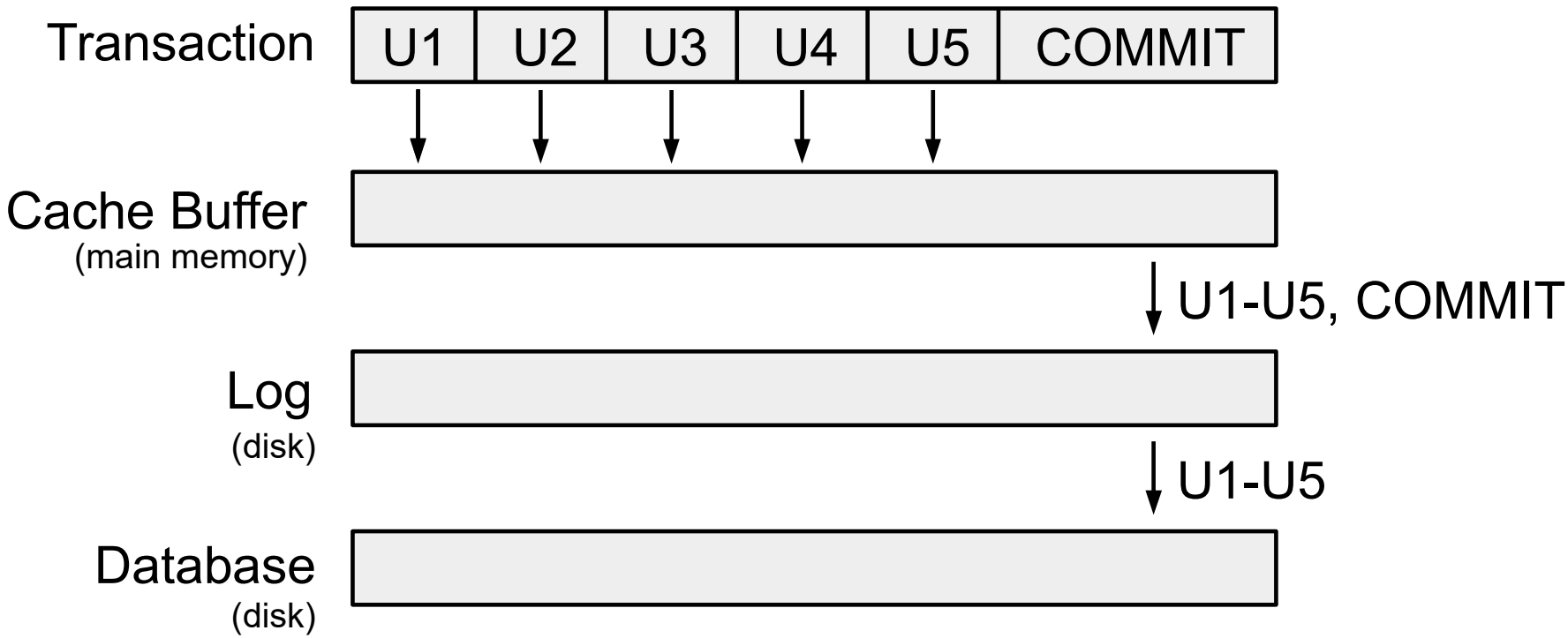
These names are related to the update of a database (data files on the secondary storage).

1 - Recovery using a deferred update 1/3



- **Deferred update** - we do not write any transaction updates in a database until we commit the transaction. Updates are simply stored in the cache buffer.
- When a transaction reaches its commit point, we first store the updates into the log file and than (when it is suitable) the updates are written in the database.
- We **do not process UNDO** in the case of a failure (the database is not updated).
- **REDO** is processed in the case of a global failure when the updates are not written in the database.
- **New values** are written in log (due to REDO).

1 - Recovery using a deferred update 2/3



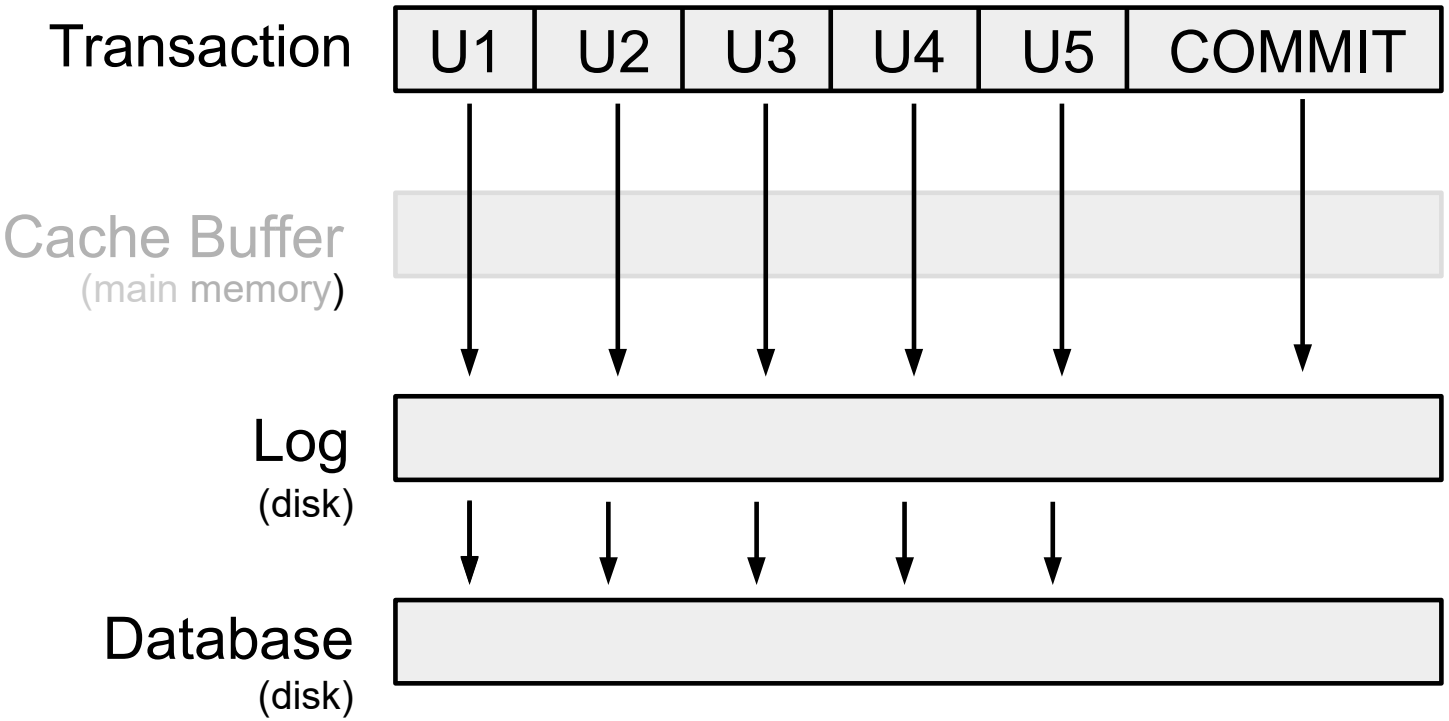
1 - Recovery using a deferred update 3/3

- This technique is called **NO-UNDO/REDO algorithm**.
- *Advantage*: A low number of I/O.
- *Disadvantage*: Possibility of buffer expansion which can lead to a low transaction throughput.
- It is usually used only in a DBMS with short transactions, now it is used in In-Memory DBMS.

2 - Recovery using an immediate update 1/3

- In this case, updates of transactions are written into a database before the commit point.
- We have to first write updates into a log file and than we immediately update the database.
- If transaction fails before the commit point we have to cancel the updates in the database using the **UNDO** operation.
- **Old values** are written into log (due to UNDO).

2 - Recovery using an immediate update 2/3



2 - Recovery using an immediate update 3/3

- If all transactions changes are written in the database before a global failure then we do not have to process the REDO operation (**UNDO/NO-REDO algorithm**).
- *Advantage*: fast recovery management, low memory requirements.
- *Disadvantage*: a high number of I/O.

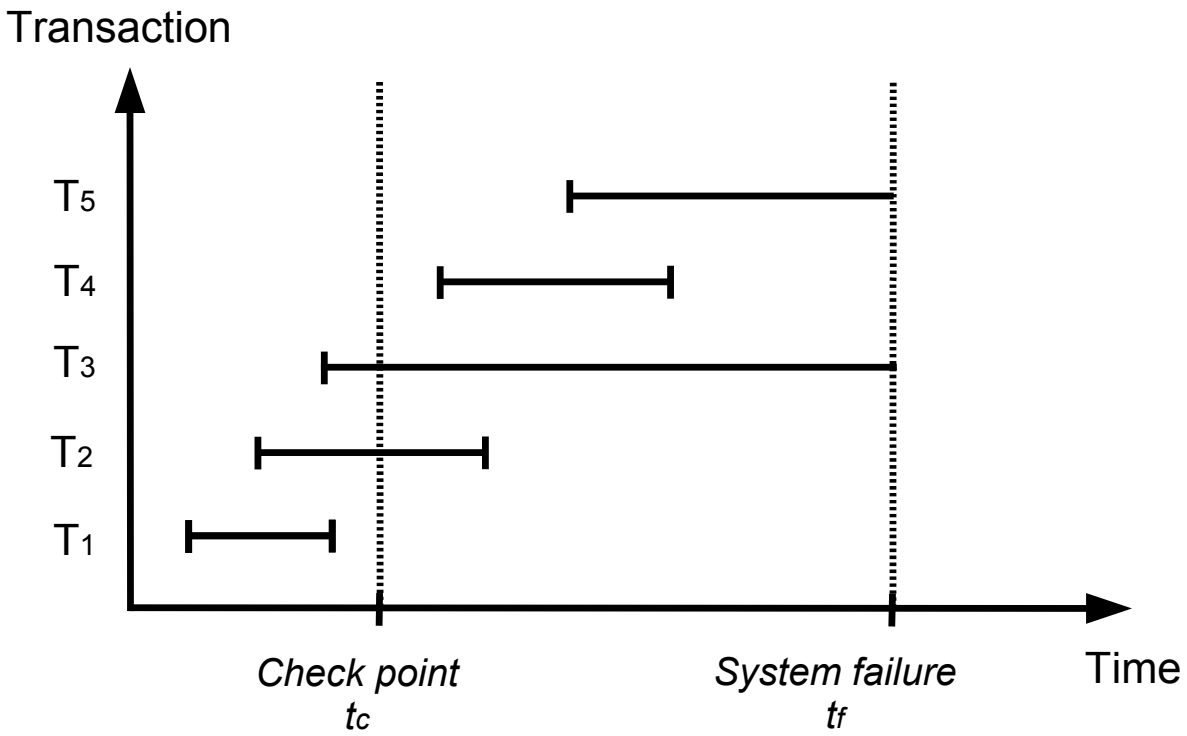
3 - A Combined Technique

- A real implementation considers a combination of both techniques.
- In this case, updates are written into the database during the transaction run (before the commit is reached).
- As a result, DBMS has to process both UNDO and REDO operations during the system recovery (the **UNDO/REDO algorithm**).
- All updates of current transactions are written in the time of the **checkpoint**.

Check points

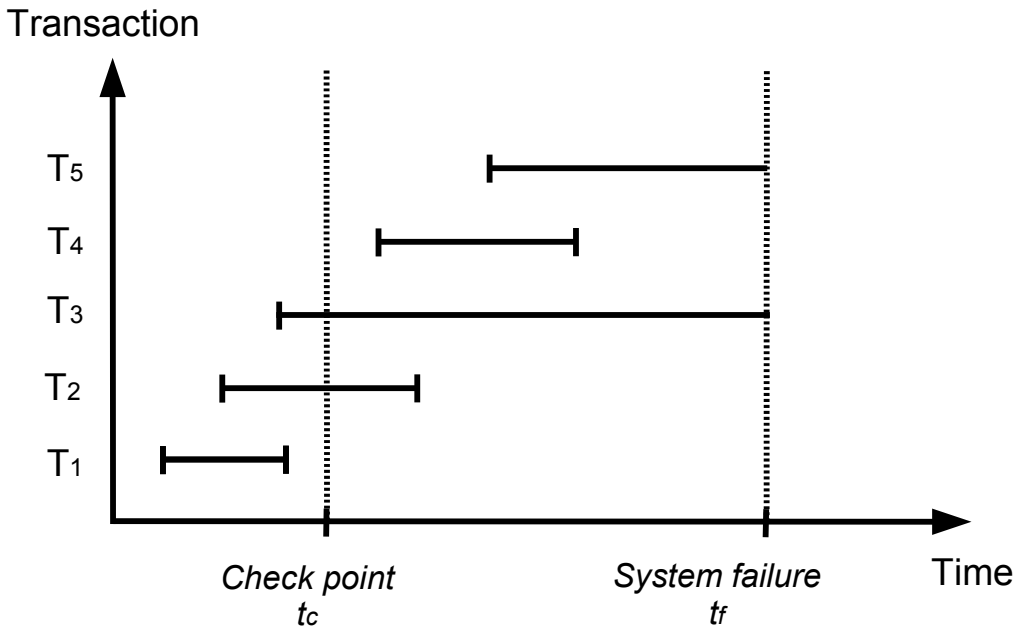
- RDBMS creates **check points** to update a database at a certain period of time.
- The check point record involves all transactions in the time of check point and all transactions committed after the previous checkpoint.
- The check point involves:
  - Write the updates into the log file and database.
  - Write the check point record into the log file.
- The updates of transactions are written into the log file:
  - When the commit of a transaction is reached.
  - Before the updates are written into the database in the time of check point.

Check points, example – initial situation 1/4



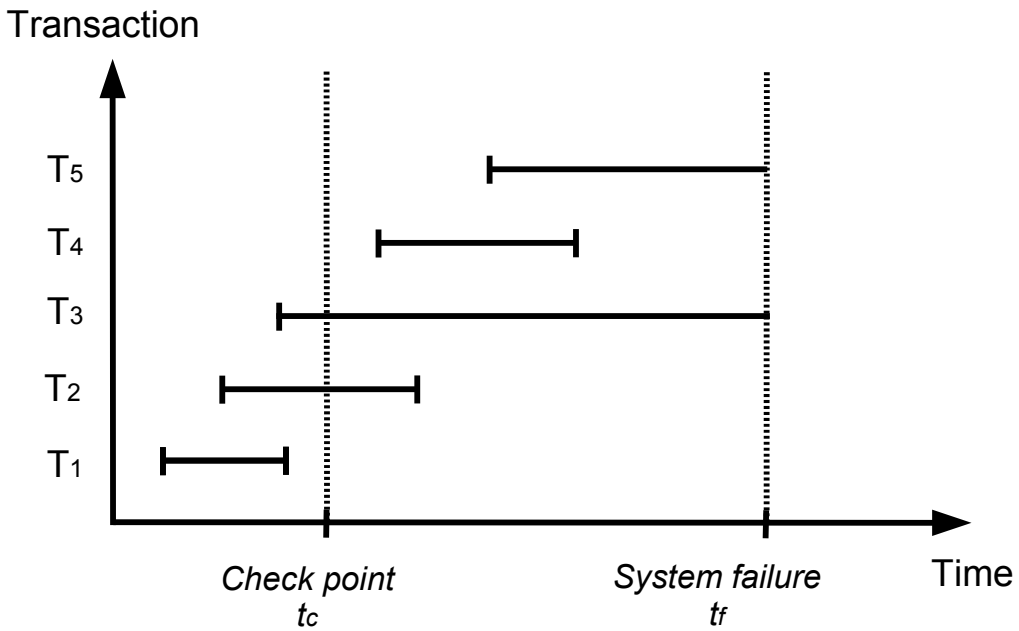
- A system failure happened in time t_f.
- A check point t_c was the last check point created before the system failure.

Check points, example – initial situation 2/4



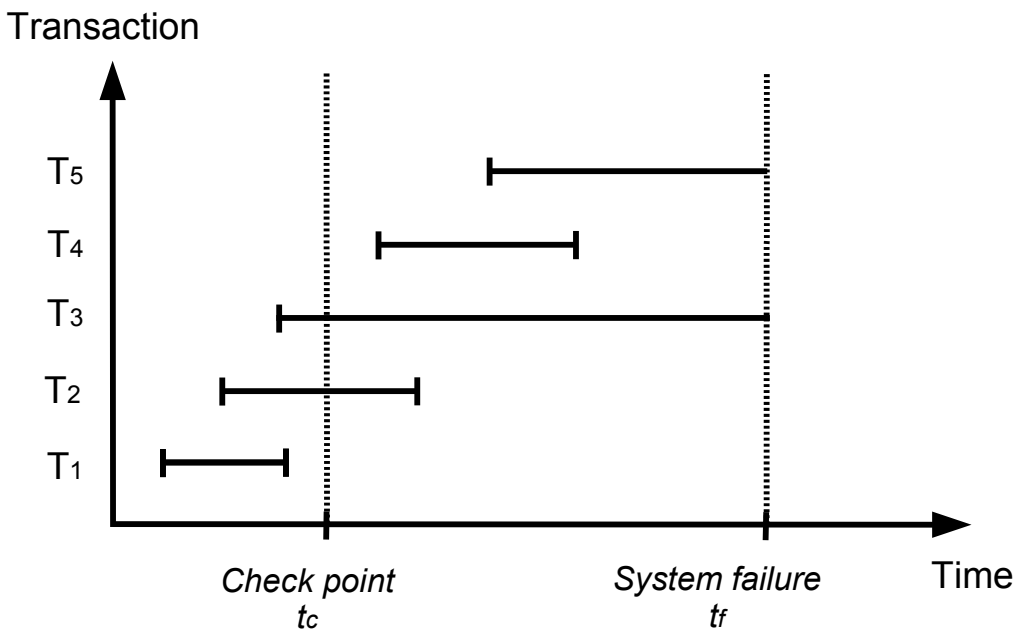
- The transaction T_1 was successfully finished before t_c, updates are written into the log file (since we would lose updates in the case of a system failure between the transaction end and t_c). The updates are not written into the database in the time of commit.

Check points, example – initial situation 3/4



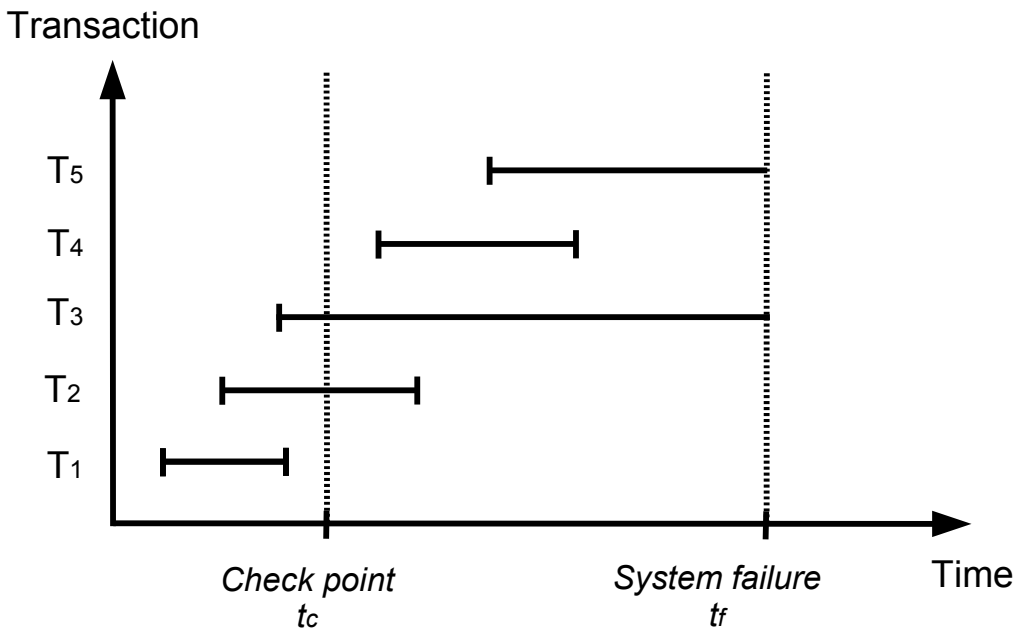
- The transaction T_2 begins before t_c and it was finished after t_c and before t_f.
- Transaction T_3 begins before t_c but it was not finished before t_f.

Check points, example – initial situation 4/4



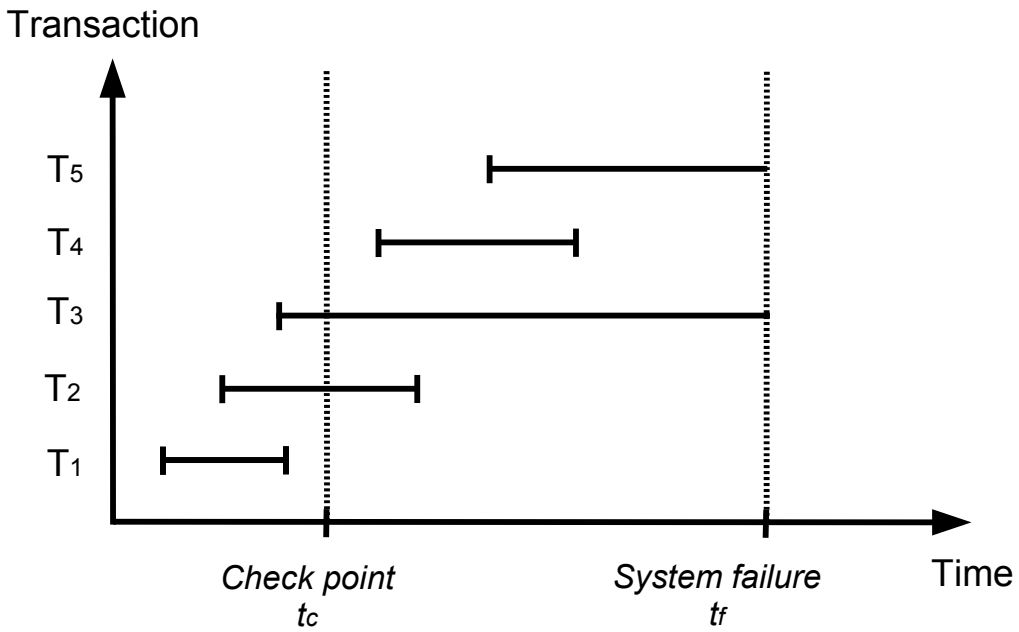
- The transaction T_4 begins after t_c and it was finished before t_f.
- Transaction T_5 begins after t_c and it was not finished before t_f.

Check points, example – Check point



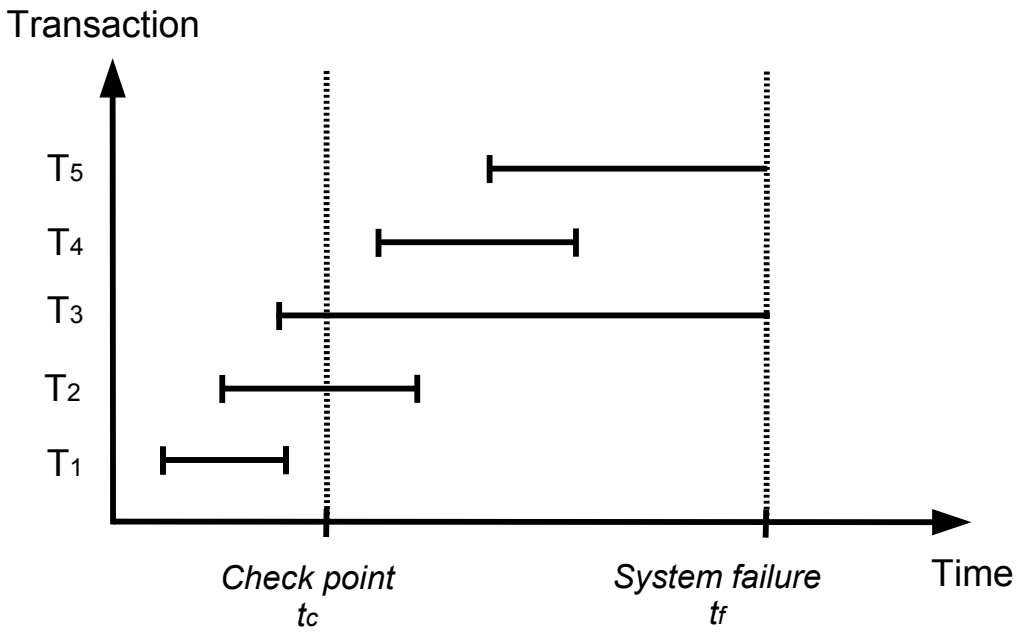
- In the time of the check point t_c:
  - All updates of T_1 are written into the database.
  - A part of updates of T_2 and T_3 (executed before t_c) are written into the database.
  - Since T_4 and T_5 have been started after t_c, no updates are written.

Check points, example – System Recovery 1/2



- T_5 is automatically cancelled since all updates have been written only into the cache buffer and it is lost.
- T_3 is cancelled – updates of T_3 written into the database in the time of t_c have to be **undo** from the database using old values written in the log file.

Check points, example – System Recovery 2/2



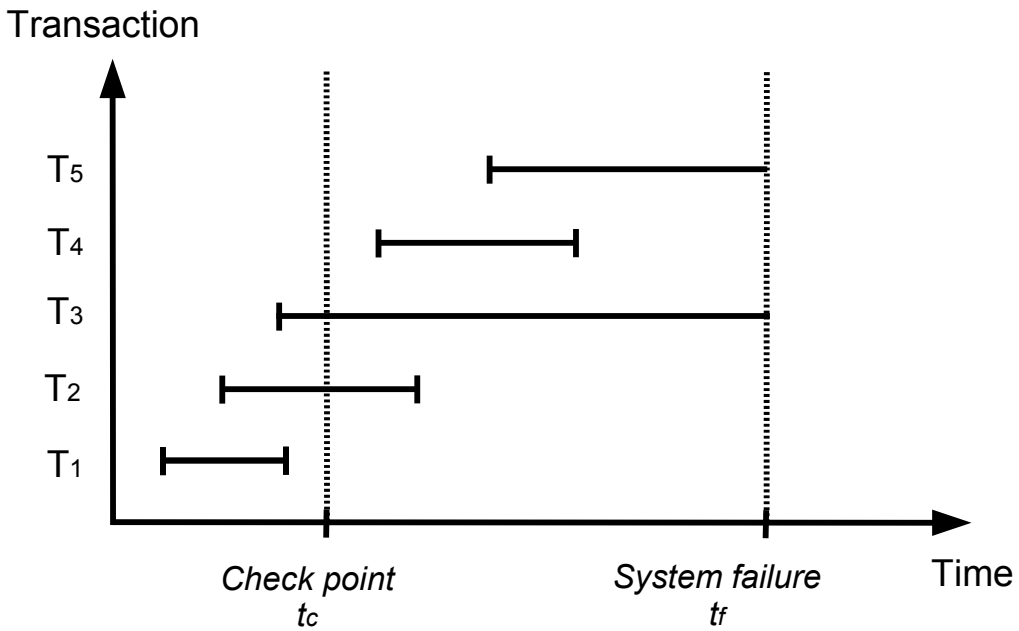
- All updates of T_4 and updates of T_2 executed after t_c are **redo** using the new values written into the log file.
- We do not consider T_1 since it was finished before t_f, all updates have to be written into the database in the time of t_c.

Recovery, UNDO/REDO

We apply the following algorithm:

1. Create two lists of transactions: UNDO a REDO.
2. Add all transactions into the UNDO list which were not be handled by the last check point. The REDO list is empty.
3. Iterate log records from the last check point
   1. If we find BEGIN TRANSACTION of the transaction T, T remains in the UNDO list.
   2. If we find COMMIT of the transaction T, move T from the UNDO list into the REDO list.

Check points, example 4/4



After the algorithm is processed, the UNDO list contains T_3 and T_5 and the REDO list includes T_2 and T_4. After that, the system traverses the log backwards and cancel operations of transactions from the UNDO list. Than system traverses the log file forward and apply operations of transaction from the REDO list.

Hardware failure

- Recovery in the case hardware failure starts with the database recovery from the back-up copy (database dump).
- We recover the database using a log file, where we **redo** all transactions finished after the creation of the back-up copy.
- We **do not undo** any operations since these changes were cancelled by lost of the database.
- There can be two problems:
  - There is not a dumb file. (It is a task for a database administrator.)
  - The log file is lost. (It is a task for RAID of a disk array.)

Save points

- We considered a transaction as an inseparable sequence of operations.
- There is a concept of *save points* introduced in SQL99 which separates a transaction to smaller parts.
- In the case of the ROLLBACK operation, we do not cancel the whole transaction but we return to the last successful **save point**.
- Save point **is not equivalent** to the commit point since the changes are not visible for other transactions until the transaction is committed (due to isolation).

Transactions in SQL

- Transactions in SQL follow the theory described in previous slides.
- All SQL commands are atomic[1].
- Operation BEGIN TRANSACTION is processed by a command `START TRANSACTION` in SQL.
- Operation COMMIT is processed by a command `COMMIT WORK`.
- Operation ROLLBACK is processed by a command `ROLLBACK WORK`.

[1] with exception of CALL and RETURN

START TRANSACTION Syntax 1/2

```
START TRANSACTION <optional parameters>;
```

Where the *<optional parameters>* specify *access mode*, *isolation level* and *diagnostics area size*.

START TRANSACTION Syntax 2/2

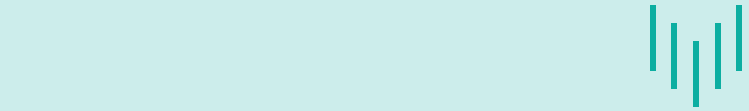
- Isolation level has the following format: `ISOLATION LEVEL` <*isolation*>, where <*isolation*> is `READ UNCOMMITED`, `READ COMMITED`, `REPEATABLE READ` a `SERIALIZABLE` (see other lectures).
- Access mode can be: `READ ONLY` or `READ WRITE`. Without any specification of isolation level, `READ WRITE` is the default value. With `READ UNCOMMITED` isolation level, `READ ONLY` is the default value, and with `READ UNCOMMITED` isolation level, `READ ONLY` is the default value. If we use `READ WRITE` isolation level, than we can not set `READ UNCOMMITED` as an access mode.
- The size of diagnostics area sets a number of exceptions stored by system on a stack. We provide this number after the `DIAGNOSTICS SIZE` keyword.

COMMIT a ROLLBACK syntax

```
COMMIT [WORK] [AND [NO] CHAIN];
ROLLBACK [WORK] [AND [NO] CHAIN];
```

- `WORK` is a complementary word and `AND NO CHAIN` is implicit.
- `AND CHAIN` automatically proceed `START TRANSACTION` with the same parameters as in the previous example after `COMMIT`.
- All cursors are closed automatically when the transaction is finished.

Save points in SQL



- Save point is created by a command:
  `SAVEPOINT` <*save point name*>;
- Command `ROLLBACK TO` <*save point name*>; cancels all operations executed after the specified save point.
- Command `RELEASE` <*save point name*>; cancels the specified save point. Therefore, we can not ROLLBACK to this save point.
- All save points are removed after the transaction is processed.

Transaction in PL/SQL, Example I, 1/2

Let us consider the table Student:

```
CREATE TABLE Student (
  login CHAR(5) PRIMARY KEY,
  fname VARCHAR(30) NOT NULL,
  lname VARCHAR(30) NOT NULL,
  email VARCHAR(40) NOT NULL
);
```

Transaction in PL/SQL, Example I, 2/2

```
BEGIN
  INSERT INTO student VALUES('sob28', 'Jan', 'Sobota',
    'jan.sobota@vsb.cz');
  INSERT INTO student VALUES('sob28', 'Jan', 'Neděle',
    'jan.nedele@vsb.cz');
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

Transaction in PL/SQL, Example II, 1/3

Let us have a database of authors and reviewers of articles. One person can have many roles (author, reviewer, administrator and so on).

```
CREATE TABLE Person (
  login          VARCHAR(20) PRIMARY KEY,
  email          VARCHAR(50) UNIQUE NOT NULL,
  password       VARCHAR(20) NOT NULL,
  firstName      VARCHAR(20) NOT NULL,
  middleName     VARCHAR(20),
  secondName     VARCHAR(20) NOT NULL,
  email2         VARCHAR(50),
  web            VARCHAR(70));
```

Transaction in PL/SQL, Example II, 2/3

Table `Role`:

```
CREATE TABLE Role (
  id              INT NOT NULL PRIMARY KEY IDENTITY,
  name            VARCHAR(50) NOT NULL UNIQUE);
```

and the table recording roles of persons:

```
CREATE TABLE PersonRole (
  idPerson        INT REFERENCES Person NOT NULL,
  idRole          INT REFERENCES Role NOT NULL,
  UNIQUE(idPerson, idRole));
```

Transaction in PL/SQL, Example II, 3/3

When a new person is inserted, we want to insert a role "Autor"(with `id=1`). The transaction is as follows:

```
BEGIN
  INSERT INTO Person VALUES('sob28', 'jan.sobota@vsb.cz',
    'heslo', 'Jan', NULL, 'Sobota', NULL, NULL);
  INSERT INTO PersonRole('son28', 1);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

If insertion of the person fails, e.g. the person is already written in the database, insertion of the role is not correct and the whole transaction is canceled.