Databases

Active Databases

Danilo Montesi

danilo.montesi@unibo.it

Passive Databases

- Reaction strategies in the integrity constraints are the first example of the need to introduce a **reactive** behaviour in the databases:

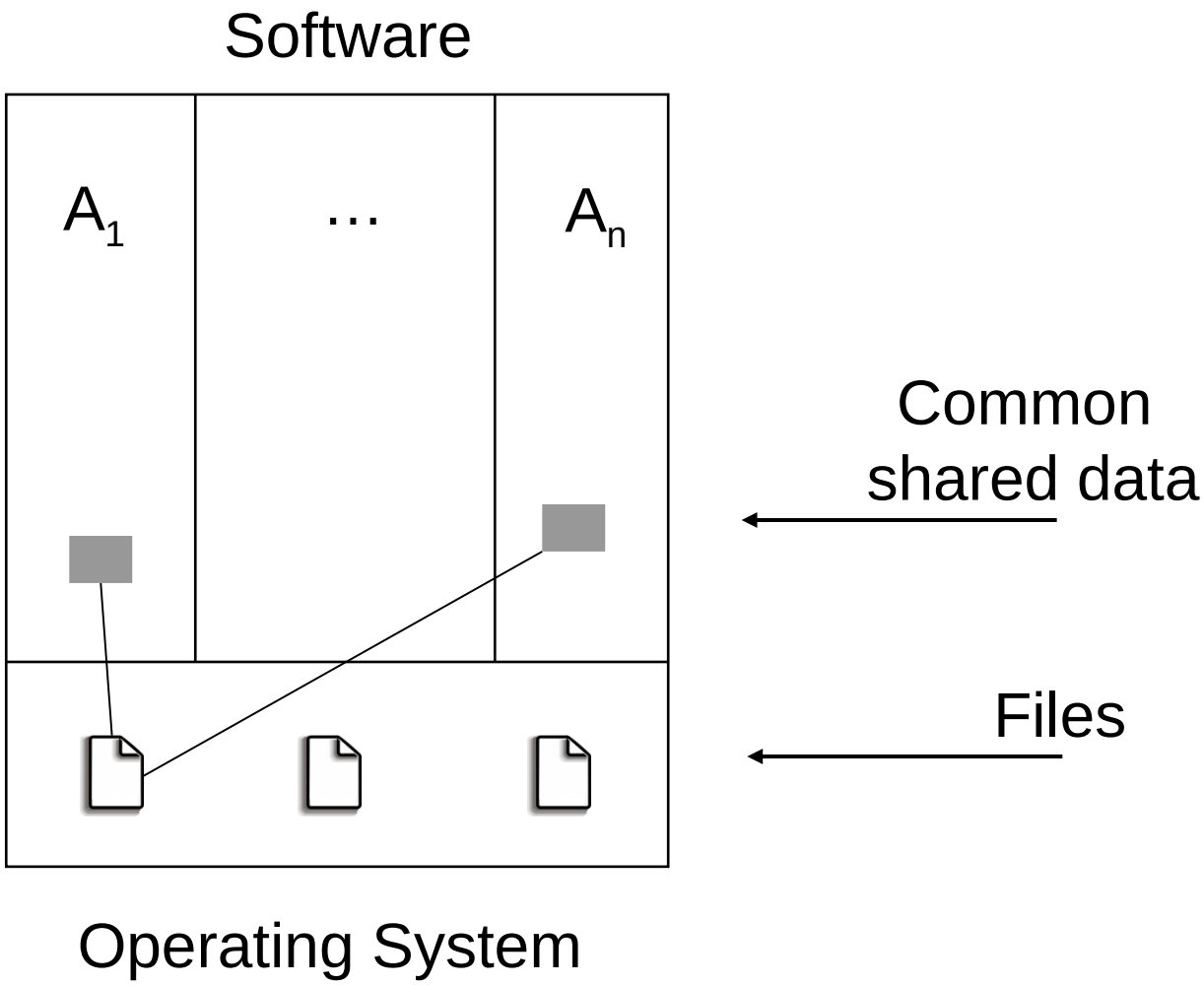
```
on   < delete | updated >
     < cascade | set null | set
       default | no action >
```

- The idea is to introduce **language constructs** specific for this goal
- These constructs are called (active) **rules** to handle a part of the procedural behaviour of an application
- Being at the database level, this behaviour is therefore “shared” among many applications, achieving data independence

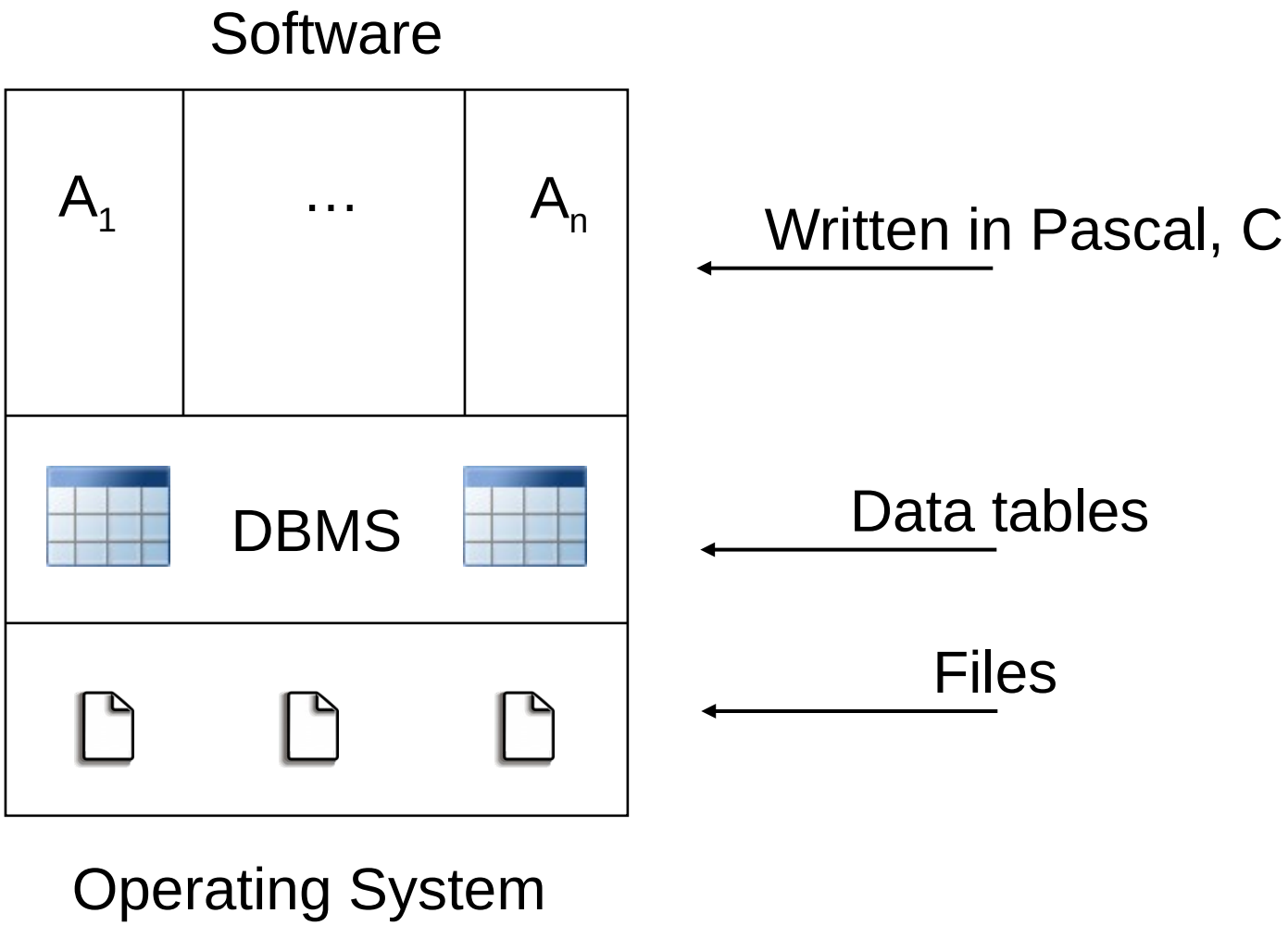
Active Databases

- An active database has a component to handle active rules of the kind **Event-Condition-Action**:
  - events are changes in the database
  - a condition is verified, based on a true/false value
  - one or more actions are executed
- These databases have a **reactive** behaviour (opposite to **passive**): they do not just execute the user transactions, but also the rules
- Commercial DBMSs (SQL3) use **triggers** to specify rules, like the ones in the schema definition

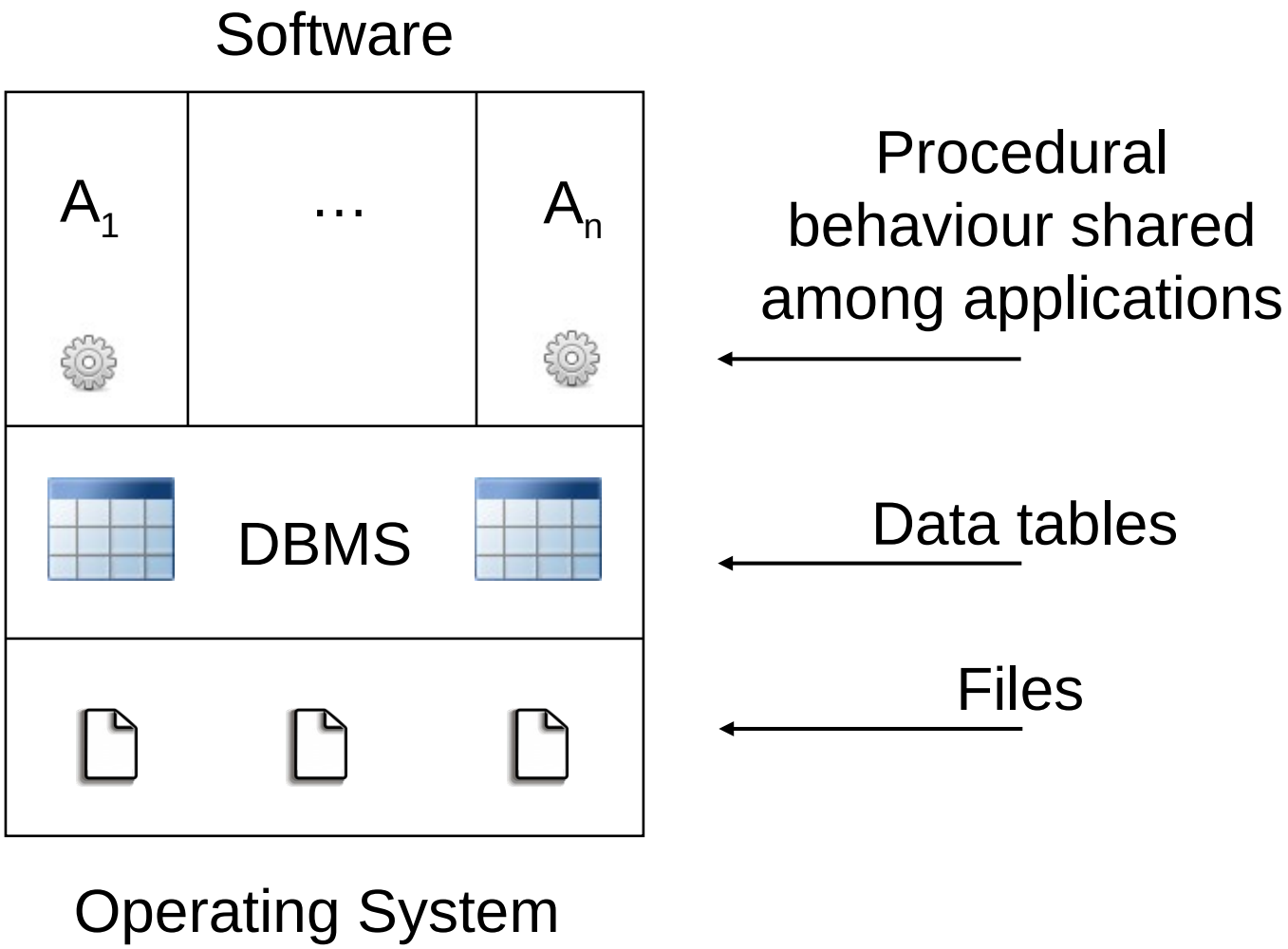
The 70s: no DBMS



The 80s: First DBMSs



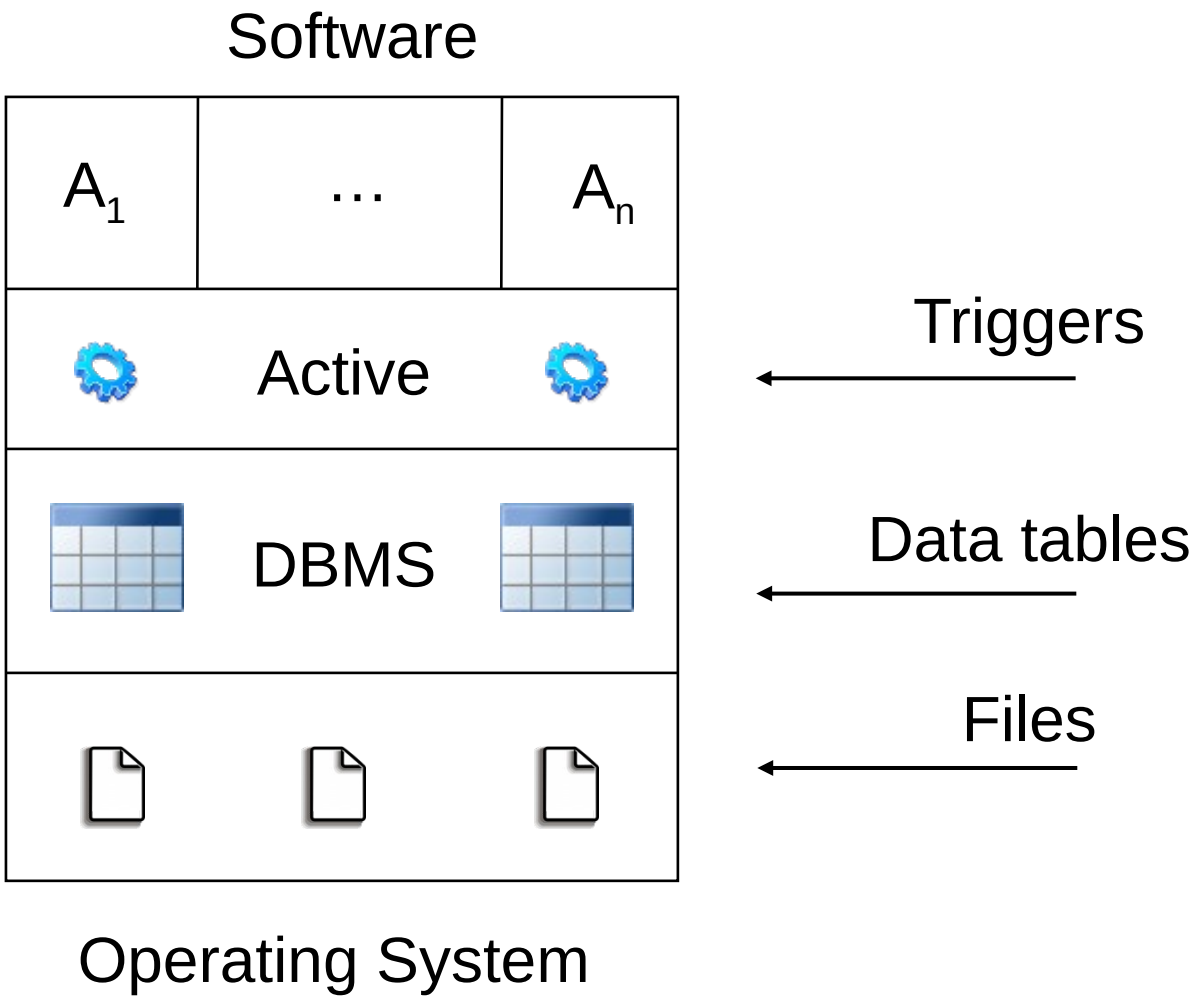
The 90s: the Procedural Behaviour



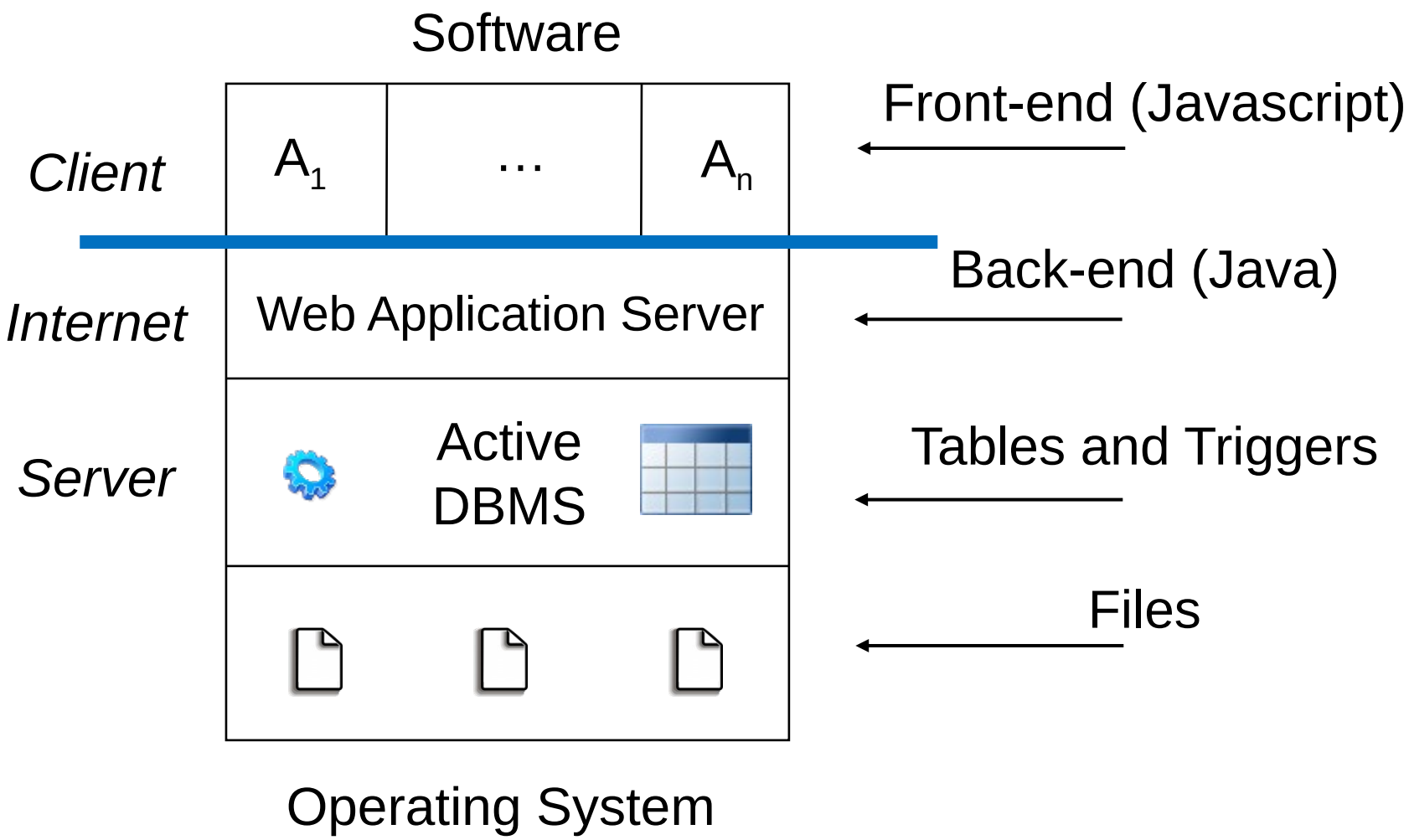
Advances in DBMSs: Stored Procedures

- Stored procedures have been introduced to share the common procedural behaviours between different software
- Stored procedures are not standardized and are affected by the problem of impedance mismatch with the language used to express such procedures
- As a result specific rules (*triggers*) have been introduced to model the procedural behaviour shared among different software and are handled by the DBMS itself

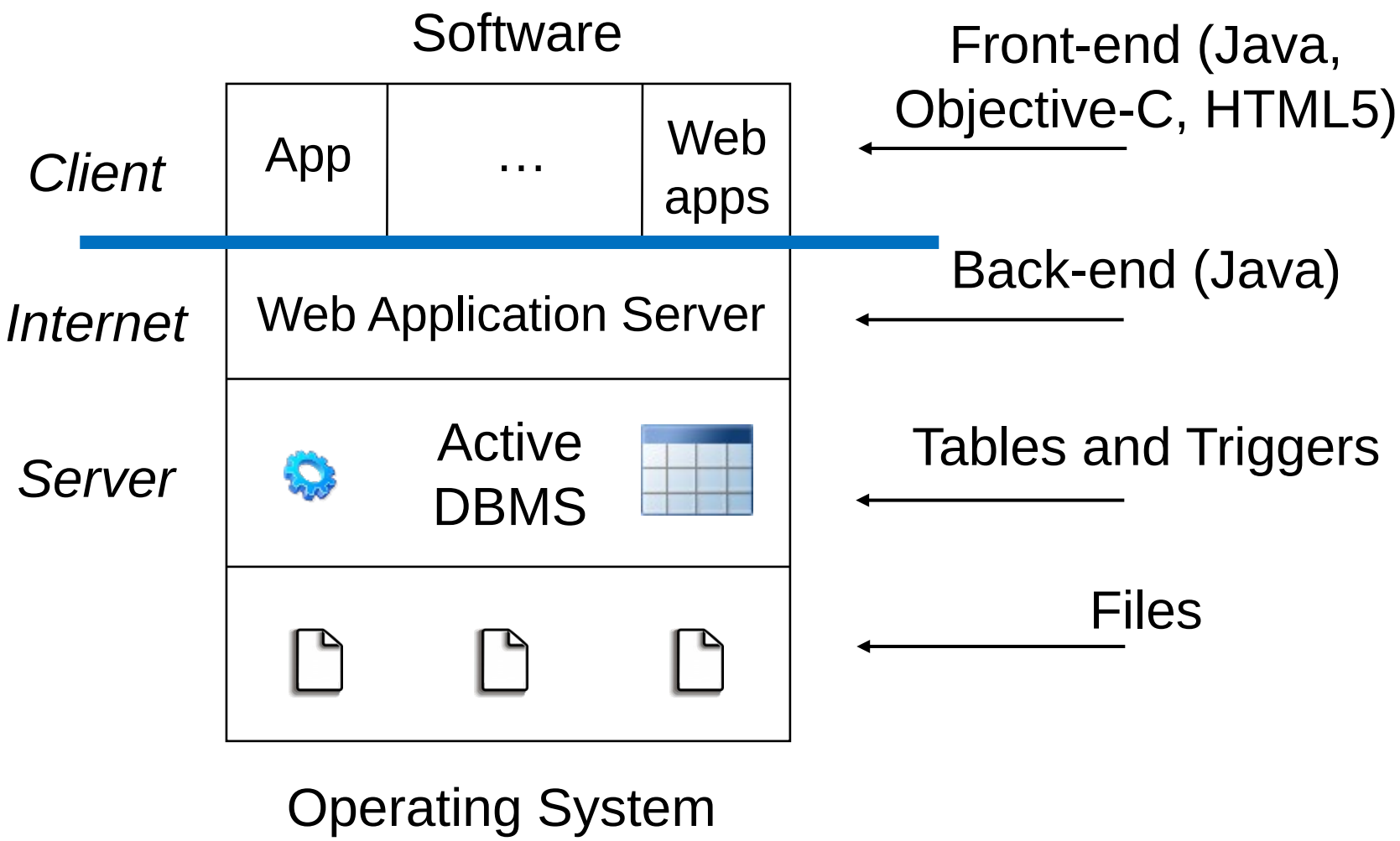
The 90s: Active DBMS



Advance in the 2000s



The 2010s: Mobile Apps



Active Database

- A database that contains active rules (called triggers)
  - Definition of triggers in SQL:1999
  - Definition of triggers in Oracle and DB2
  - Design issues for applications based on the use of triggers

Trigger

- Defined with DDL instructions (`create trigger`)
  - based on the event-condition-action (ECA) paradigm:
    - event: change in the data, specified using `insert`, `delete`, `update`
    - condition: (optional) SQL predicate
    - action: sequence of SQL instructions (or extensions, for example PL/SQL in Oracle)
  - intuitively:
    - when there is an event (activation)
    - if the condition is satisfied (verification)
    - then execute the action (execution)
  - each trigger refers to a (target) table: it responds to events related to that table

Trigger: Granularities and Modalities

- Granularity
  - of tuple (**row-level**): activation for each tuple involved in the operation
  - of operation (**statement-level**): only one activation for an SQL instruction, with reference to all the tuples involved ("set-oriented")
- Mode
  - **immediate**: right after (or right before) of the event
  - **deferred**: at the time of commit

Computational Model

- Let $T^U = U_1; \ldots ; U_n$ the user transaction
- If the rules of P have form E, C → A with E event, C condition and A action, then:
  - **Immediate** semantic generates
    $$T^I = U_1; \underline{U^P_1}; \ldots U_n; \underline{U^P_n}$$
  - **Deferred** semantic generates
    $$T^D = U_1; \ldots U_n; \underline{U^P_1}; \ldots ; \underline{U^P_n}$$
- Where $\underline{U^P_i}$ represents the sequence of actions induced by U_i on P
- Problems:
  - Termination
  - Confluence
  - Equivalence

Trigger in SQL:1999: Syntax

```
create trigger triggerName
  { before | after }
  { insert | delete | update [of
  column] }
    on targetTable
  [referencing
    {[old table [as] oldTableVar]
    [new table [as] newTableVar] } |
    {[old [row] [as] oldTupleVar]
    [new [row] [as] newTupleVar] }]
  [for each { row | statement }]
  [when condition]
  SQLProceduralStatement
```

Types of Events

- **Before**
  - The trigger is considered and possibly executed before the event
  - It is usually used when you want to verify a change before it occurs and “modify the change”
- **After**
  - The trigger is considered and executed after the event
  - This is the most common mode, suitable for most applications

Referencing Clause

- Depends on granularity
  - statement-level mode: two transition tables (**`old table`** and **`new table`**) contain the previous and next values of the tuples modified by the statement
  - row-level mode: two transition variables (**`old`** and **`new`**) represent the value before or after a tuple is modified
- **`old`** and **`old table`** are not present with the insert event
- **`new`** and **`new table`** are not present with the delete event

Trigger in Oracle: Syntax

```
create trigger triggerName
  { before | after } event [, event [, event]]
  [[referencing
    [old [row] [as] oldTupleVar]
    [new [row] [as] newTupleVar]]
  for each { row | statement } [when condition]]
  PL/SQLBlock
```

- { before | after }: event type
- *event:* insert, update, delete
- for each row specifies the granularity
- *Reference:* it allows to define variable names (usable only for tuple granularity)

  old as *OldVariable* | new as *NewVariable*

Trigger in Oracle: Semantic

- **Immediate** mode, both `after` and `before`
- Execution plan
  - `before statement` trigger
  - for each tuple involved
    - `before row` trigger
    - execution and constraints check
    - `after row` trigger
  - `after statement` trigger
- In the event of an error, everything is discarded
- Priorities between triggers, using timestamp
- Max 32 triggers activated in cascade

Trigger in Oracle: an Example (1)

```
create trigger Reorder
   after update of QtyAvbl on Warehouse
   when (new.QtyAvbl < new.QtyLimit)
   for each row
      declare X number;
   begin
      select count(*) into X
      from PendingOrders
      where Part = new.Part;
      if X = 0
      then
            insert into PendingOrders
            values (new.Part, new.QtyReord,
sysdate);
      end if;
   end;
```

Trigger in Oracle: an Example (2)

Warehouse

Part	QtyAvbl	QtyLimit	QtyReord
1	200	150	100
2	780	500	200
3	450	400	120

```
T1: update Warehouse
    set QtyAvbl = QtyAvbl -
70
    where Part = 1

T2: update Warehouse
    set QtyAvbl = QtyAvbl -
60
    where Part <= 3
```

Trigger in DB2: Syntax

```
create trigger triggerName
  { before | after } event on
  targetTable
  [referencing reference]
  for each level
  [when (SQLPredicate)]
  SQLProceduralStatement
```

- { **before** | **after** }: event type
- *event:* insert, update, delete
- for each *level* specifies the granularity
- *Reference:* it allows to define variable names (depending on granularity):

```
  old as OldTupleVar | new as NewTupleVar
old_table as OldTableVar | new_table as NewTableVar
```

Trigger in DB2: Semantic

- **Immediate** mode, both `after` and `before`
  - `before` triggers cannot modify the database, apart from variants on the changes caused by the event (therefore they cannot in general activate other triggers)
- In the event of an error, everything is discarded
- No priority between triggers (the order is defined by the system, *i.e., timestamp*), interaction with compensatory actions on referential integrity constraints
- Max 16 triggers activated in cascade

Trigger in DB2: an Example

```
create trigger checkWage
  after update of Wage on Employee
  for each row
  when (new.Wage < old.Wage * 0.97)
  begin
    update Employee
    set Wage = old.Wage * 0.97
    where EmpCode = new.EmpCode;
  end;
```

Extensions (not usually available)

- Temporal events (also periodical) or “user-defined”
- Boolean combinations of events
- Clause `instead of`: it does not execute the operation that activated the event, but another action instead
- “Detached” execution: an autonomous transaction is activated
- Priorities definition
- Groups rules, can be activated and deactivated
- Rules associated also with queries (not just updates)

Rules Properties

- Termination (essential)
- Confluence
- Determinism of observations

Applications

- Internal features
  - Handling of integrity constraints
  - Replication
  - View management
    - Materialized: propagation
    - Virtual: modification of the queries
- Application features: description of the behaviour of the database