Exercise Sheet: Denormalization and Aggregates

Data, Data Storage, Data Collection

Exercise 1 – A Library Network

The public libraries of a city share one database, built from the following rules.

- Each member is a registered person with an identifier, a name and an address.
- A book has an ISBN, a title, and one or more authors. An author may write several books.
- The network has ten branches. Each branch has a name, unique in the city, and an address.
- The network holds several copies of a book. Each copy has an identifier and belongs to one branch.
- A loan is one copy lent to one member on one date. It records the date the copy is returned, left empty while the copy is on loan. A copy is lent at most once a day.

(**Q1**) Design a schema from these rules: give its tables, underline their keys, and name the foreign keys.

Three rules need a decision.

- A book has several authors and an author several books, so the authors cannot be an attribute of Books. Authors get a table of their own, with an identifier, since two authors may share a name, and a pair (book, author) gets a table of its own, with both attributes as its key.
- A branch has an address of its own, so branches get a table, and a copy stores only the name of its branch. Storing the address in each copy would repeat it once per copy.
- A copy is lent at most once a day, so a copy and a date identify a loan: $\{\text{CopyID}, \text{LoanDate}\}$ is a key. Giving each loan an identifier of its own is also correct.

- Members(<u>MemberID</u>, Name, Address)
- Books(<u>ISBN</u>, Title)
- Authors(<u>AuthorID</u>, Name)
- BookAuthors(<u>ISBN</u>, <u>AuthorID</u>), where ISBN references Books and AuthorID references Authors.
- Branches(<u>Branch</u>, BranchAddress)
- Copies(<u>CopyID</u>, ISBN, Branch), where ISBN references Books and Branch references Branches.
- Loans(<u>CopyID</u>, <u>LoanDate</u>, MemberID, ReturnDate), where CopyID references Copies and MemberID references Members.

In each table, every non-trivial dependency has the key on its left, so every table is in BCNF. Splitting Name into FirstName and LastName is also correct: it changes no key and no normal form.

Every September, the city's schools send the library a flat file with schema Pupils(PupilName, Address, School, Class), with one row per pupil to give a library card, and no member identifier. Some of these pupils are already members.

(**Q2**) Where would you store this file, and how would you connect its rows to the members already stored?

The file is stored as it arrives, in its own table, and its rows are connected to Members in a separate table:

- PupilsRaw(RecordID, PupilName, Address, School, Class) holds one row per row of the file. The file has no key of its own, so the library gives each row a number, RecordID, on loading.
- Matches(RecordID, MemberID) holds one row per pupil connected to a member, found by comparing names and addresses, by a rule or by hand.

Matching the rows while loading them into Members would fail whenever the same person is written differently in the two sources (a typo, "12 rue" against "12, rue", a family that moved): an exact match misses the pupil, a looser one may connect the wrong member. Keeping the raw rows means never editing the raw data: a wrong match is undone by deleting its row in Matches, and the matching can be run again from the original rows.

(**Q3**) Which data of the school file, once stored, is redundant with the rest of the database? Why is this denormalization not a choice?

For every pupil who is already a member, PupilName and Address repeat the name and address stored in Members. The library does not control the schools' format and has no identifier to connect the rows reliably, so it has to keep the data as it comes: this denormalization is a constraint of data integration. If the schools could look up the MemberID of each pupil, the file would no longer need names and addresses for members: Enrollments(MemberID, Year, School, Class), with new pupils first registered in Members, stores no duplicate. The redundancy comes from the missing identifier, not from a choice.

The library stores the copies and the loans in the tables

- Copies(CopyID, ISBN, Branch) and
- Loans(CopyID, LoanDate, MemberID, ReturnDate).

At the moment, each copy stays in its branch: a copy borrowed at one branch is brought back to it. The network plans to let copies float: a copy returned at another branch would stay there, and Copies would then record its new branch. The statistics of each branch must count the loans made there, so the designers already add a column Branch to Loans, filled at each loan with the branch of the copy.

(**Q4**) Under today's rule, show that this column is redundant: which dependency holds in the new Loans table, and which normal form does it violate?

Under today's rule, a copy never changes branch, so all the loans of one copy carry the same branch, and $\{\text{CopyID}\} \to \{\text{Branch}\}$ holds in Loans. The key of Loans is $\{\text{CopyID}, \text{LoanDate}\}$, and Branch depends on CopyID alone, a proper part of it, with Branch not prime: the dependency is partial. Loans is therefore not in 2NF, nor in any higher normal form. The column Branch of Loans is redundant: the branch of a copy is stored once in Copies and once more for each of its loans.

(**Q5**) The plan is adopted. Copy C7 belongs to the North branch, is lent twice there, and is then returned at the South branch, where it is lent once more:

CopyID	LoanDate	Branch
C7	2 Sep	North
C7	20 Sep	North
C7	9 Oct	South

(Columns MemberID and ReturnDate not shown.) Does {CopyID} → {Branch} still hold in Loans? Which normal form is the Loans table in?

No, {CopyID} → {Branch} no longer holds in Loans. Two tables now store a branch for C7, and they record two different facts:

- Copies records the branch that holds the copy: the branch of its last return, which is also the branch of the ongoing loan while the copy is out. For C7: South.
- Loans records where each loan was made: North, North, then South.

The North of the first two loans is therefore no longer a duplicate of a value stored in Copies: it is a fact about the loan. A functional dependency is a rule on every state the table may take. The loans of 2 Sep and 9 Oct have the same CopyID and different branches, so this state already breaks the dependency. Every non-trivial dependency left has the key {CopyID, LoanDate} on its left: the new Loans table is in BCNF. What was a redundant column under the old rule is the normalized design under the new one, and adding the column in advance meant that the change needed no migration.

The library stores the books and the branches in the tables

- Books(ISBN, Title) and
- Branches(Branch, BranchAddress).

The network holds 200 000 copies and has recorded about two million loans over the years, with about 2 000 loans and returns a day. Its catalogue search sends about 100 000 queries a day. Each result shows the title of a book and, for each of its copies that is not on loan, the branch and its address. In the normalized schema, each search joins Books, Copies, Branches and Loans.

(**Q6**) To avoid these joins, a colleague proposes to store Copies(CopyID, ISBN, Title, Branch, BranchAddress). Which dependencies of this table break BCNF, and what can go wrong? What should be tried before changing the tables?

In the colleague's table, {ISBN} → {Title} and {Branch} → {BranchAddress} hold, and neither ISBN nor Branch is a superkey, since a book and a branch have many copies. Both dependencies are transitive from the key CopyID, so the table is in 2NF but not in 3NF, and therefore not in BCNF. The address of a branch is stored once per copy, about 20 000 times, which may cause any of the three modification anomalies:

- **Update anomaly:** correcting the address rewrites all these rows, and a row missed leaves the branch with two addresses.
- **Insertion anomaly:** a new branch with no copy yet has no row to hold its address.
- **Deletion anomaly:** deleting the last copies of a branch also deletes its address.

The title of a book, stored once per copy, causes the same three anomalies. Before changing the tables, the physical level comes first. Possible answers include:

- An index on Copies by ISBN, which finds the copies of a book without reading the 200 000 rows.
- An index on Loans by CopyID and ReturnDate, which finds the ongoing loan of a copy, if any, without reading the two million loans.
- The join algorithm, which the optimizer chooses.
- A materialized view holding the join, which the system keeps up to date.

(**Q7**) The library's software is a hosted product: it creates an index on every key and every foreign key, and allows no other index and no materialized view. For each copy, a search still reads the loans of the copy to find whether one has an empty ReturnDate. Propose a change of the tables so that the search no longer reads Loans, and give two costs of keeping it.

A precomputed column in Copies:

Copies(<u>CopyID</u>, ISBN, Branch, OnLoan),

set to yes at each loan of the copy and to no at each return. The search reads the copies of the book and their flag, and no longer reads Loans, the largest table. Possible costs include:

- Every loan and every return now writes two tables, Loans and Copies.
- If one of these writes is lost, OnLoan contradicts the loans, and nothing in the schema detects it.

A flag read about 100 000 times a day and changed about 2 000 times a day is where this trade of faster reads for costlier writes pays. The colleague's duplicated columns pay less: Books and Branches are joined through their keys, which the product indexes.

Exercise 2 – Selling Concert Tickets

A small platform sells concert tickets. In a normalized schema, it stores its users, its events, and the seats of each event with the user who holds each one (empty when the seat is free):

Users

<u>UserID</u>	Name
U1	Nina
U2	Omar
U3	Lea

Events

<u>EventID</u>	Artist	Venue	Date
E1	The Pines	Arena Nd	5 Dec
E2	Mira Sol	Club 9	12 Dec

Seats

<u>EventID</u>	<u>Seat</u>	UserID
E1	A1	U1
E1	A2	U2
E1	A3	
E2	1	U1
E2	2	

The platform's workload consists of the following five operations.

W1: Show an event page: its artist, venue and date, and the seats still free.
W2: Show a user's "My tickets" page: for each ticket, the artist, the date and the seat.
W3: Book a free seat for a user.
W4: Change the date of an event.
W5: Once a month, count the tickets sold per venue.

Recall that an aggregate is a group of related data that has its own key, is stored as a single value that may contain lists and nested records, and is the unit in which the application reads and writes data.

(**Q1**) Propose aggregates for this workload, so that W1 and W2 each read a single aggregate. For each kind of aggregate, give its key and its content as a nested record, and write it out for E1 and for U1.

W1 reads an event with its seats, and W2 reads a user with their tickets: two different shapes, so two kinds of aggregates.

- An event aggregate has the key EventID. It holds the artist, the venue and the date, and the list of the seats, each with the UserID of its holder.
- A user aggregate has the key UserID. It holds the name and the list of the user's tickets, each with its EventID and its seat, and a copy of the artist and the date of the event, so that W2 needs no other read.

Written out for E1 and U1:

```
E1: { artist: "The Pines", venue: "Arena Nd", date: "5 Dec",
      seats: [ {seat: "A1", holder: "U1"},
               {seat: "A2", holder: "U2"},
               {seat: "A3", holder: null} ] }
```

```
U1: { name: "Nina",
      tickets: [ {event: "E1", seat: "A1",
                  artist: "The Pines", date: "5 Dec"},
                 {event: "E2", seat: "1",
                  artist: "Mira Sol", date: "12 Dec"} ] }
```

(**Q2**) Which data does your design copy? For each operation W1 to W5, does your design then read or write one aggregate, or several?

The artist and the date of each event are copied into every ticket of every user aggregate, and the holder of each seat is stored both in the event aggregate and in the user aggregate.

- W1 reads one aggregate, the event.
- W2 reads one aggregate, the user, thanks to the copies.
- W3 writes two aggregates: the seat's holder in the event, and the new ticket in the user.
- W4 writes the event and every user aggregate that holds a ticket for it: when the date of E1 changes, E1, U1 and U2 here, and thousands of users for a large concert.
- W5 reads every event aggregate, to count the seats with a holder per venue.

As for any denormalization, the copies make W2 a single read and W4 a write to many aggregates. W3 is the frequent operation that crosses an aggregate boundary. W5 reads everything, but once a month.

A variant keeps in the event only the list of free seats. W1 to W3 are unchanged: W3 removes the seat from the list. W5 then needs the number of seats of each event, stored in the event, to count the tickets sold. W4 can no longer find the holders in the event: it reads every user aggregate.

(**Q3**) Nina and Lea click on seat A3 of E1 at the same moment. Where must the state of the seat live so that only one of them gets it?

The seat's holder must live in the event aggregate E1. The two bookings are two concurrent writes to the same data, a race condition: without isolation, both may see

A3 free and both succeed. An operation on one aggregate is atomic, and stores built on aggregates usually also isolate it: two operations on the same aggregate run as if one after the other. The booking can therefore check that A3 is free and set its holder in one operation: the second booking then finds A3 taken and is refused. If the holder were stored only in the users' aggregates, the two bookings would write two different aggregates, U1 and U3, and both would succeed.

(**Q4**) A new event E3 opens its sale: 20 000 seats sold in ten minutes. What happens with one aggregate per event? With one aggregate per seat? Propose a boundary in between.

With one aggregate per event, every booking writes E3, and writes to one aggregate take turns: 20 000 bookings wait for each other, even when they are for different seats. With one aggregate per seat, bookings of different seats never wait for each other, but W1 must now read 20 000 aggregates to show the free seats. An aggregate that is too large makes unrelated writes wait; one that is too small turns a frequent operation into many reads. Possible answers include:

- One aggregate per block of seats (a row or a section of the hall): bookings in different blocks do not wait for each other, and W1 reads one aggregate per block, a few dozen instead of 20 000.
- One aggregate per price zone (floor, balcony): W1 reads only a handful of aggregates, but bookings in the same zone still wait for each other.

Exercise 3 – An ATM Cut Off From Its Bank

A bank stores each account as an aggregate, with two copies: one in the bank's data center, and one in the branch of a small town, which the town's ATM uses. The bank has one rule: the balance of an account is never below zero. Paul's account has a balance of 300. One evening, the link between the town and the data center is cut: a network partition. The ATM keeps working, but it can only reach the branch's copy; card payments go through the data center and its copy. At the ATM, Paul can check his balance, withdraw money, or deposit money.

(**Q1**) Replay the proof sketch of the CAP theorem on this bank: which write lands on which side of the cut, which read lands on the other side, and what are the read's two options?

- The partition splits the system into two sides: S_1, the data center, and S_2, the town's branch and its ATM.
- The write: a card payment of Paul's reaches S_1, and by availability the data center accepts it and lowers its copy of the balance.
- Since the network is partitioned, the payment cannot reach the branch's copy, which still reads 300.
- The read: Paul checks his balance at the ATM, on S_2.

The read has two options: the ATM answers 300, which misses the payment, so the system is available but not consistent; or it refuses to answer, so it is consistent but not available.

(**Q2**) Which operations at the ATM can break the bank's rule if the ATM answers during the partition?

If the ATM answers, it acts on the branch's copy only, which may miss operations made on the data center's side.

- Checking the balance shows a value that may be out of date, but it changes

nothing, so it breaks no rule.

- A deposit only adds money, so whatever happened on the other side, it cannot push the balance below zero.
- A withdrawal checks a balance that may miss payments made on the other side: the money may already have been spent, and the withdrawal can push the balance below zero.

Only withdrawals can break the bank's rule.

(**Q3**) During the partition, Paul withdraws 200 at the ATM, while a card payment of 200 goes through the data center. What is the balance once the two copies are brought together? Why did neither side refuse its operation?

Once both operations are applied to the account:

$$300 - 200 - 200 = -100.$$

The balance is below zero and the rule is broken. Neither side refused because each checked the rule on its own copy, found 300, and could not see the other side's write: two concurrent writes on two copies of one account.

(**Q4**) To stay useful during a partition, ATMs keep accepting withdrawals up to a total of about 200, and refuse beyond it. What does a higher limit buy, and what does it cost? How can the bank deal with a balance below zero after the partition?

A higher limit keeps the ATM useful to more customers during a partition: more availability. It costs risk: as in (**Q3**), a balance can end below zero. The bank accepts this because it can repair the damage afterwards: once the copies are brought together, it detects the negative balance, charges a fee and expects repayment.[a] The bank's correctness rests on detecting and compensating mistakes, not on refusing every risky operation.

[a] E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed", *IEEE Computer* 45(2), 2012.

(**Q5**) Is this ATM consistent or available during a partition?

The ATM is neither consistent nor available as a whole. Balance checks and deposits stay available, and so do withdrawals up to the limit; beyond the limit, withdrawals are refused, which keeps the risk bounded. The choice between consistency and availability is made operation by operation, and here even amount by amount.

(**Q6**) The bank's rule "a balance is never below zero" is also called a consistency rule. Is it the consistency of the CAP theorem, of ACID transactions, or of data quality?

It is ACID's consistency: a transaction takes the database from one valid state to another, here a state where no balance is below zero. CAP's consistency is about reads: every read returns the latest write, as if there were a single copy. The two are different: during the partition, Paul's balance check returned 300 although a payment had been made, which breaks CAP's consistency without breaking the rule; and a system that refuses that read keeps CAP's consistency but must still check the rule at each withdrawal. Data quality's consistency, the agreement between data points that should match, is a third notion.

Exercise 4 – Placing the Conversations on Nodes

A messaging app stores its data as aggregates: one per conversation, holding its members and its messages, and one per user, holding the user's profile, the list of their conversations and their count of unread messages. It has four conversations, C1 to C4, and four users, U1 to U4. Its data is sharded over three nodes, numbered 0, 1 and 2: each aggregate is placed on node $hash(key)$ mod 3. The hash values of the eight keys are:

key	C1	C2	C3	C4	U1	U2	U3	U4
$hash(key)$	17	22	9	32	26	13	30	10

Conversations C2, C3 and C4 were all created on 12 Dec.

(**Q1**) Place each aggregate on its node. How many nodes does showing conversation C2 ask? And listing all the conversations created on 12 Dec?

key	C1	C2	C3	C4	U1	U2	U3	U4
$hash$ mod 3	2	1	0	2	2	1	0	1

Node 0 holds C3 and U3, node 1 holds C2, U2 and U4, and node 2 holds C1, C4 and U1. Conversation C2 is one aggregate, so showing it asks one node, node 1. The conversations created on 12 Dec are C2, C3 and C4, here on nodes 1, 0 and 2. The node is computed from the key, not from the date, so without an index on dates the app must ask every node, even if the three conversations happened to share one. Hashing finds a key, but it scatters keys that are close in another sense, here the date.

To survive a failure, the app also keeps a copy of each aggregate on the next node, $(hash(key) + 1)$ mod 3.

(**Q2**) Node 1 fails. Can every aggregate still be read? And if nodes 1 and 2 fail together?

With node 1 down, every aggregate can still be read: each aggregate is on two different nodes, its own and the next one, so a single failure leaves one copy of each. With nodes 1 and 2 down, C2, U2 and U4 can no longer be read: an aggregate is lost when both its copies are on these nodes, that is when its own node is 1 and its copy on 2.

(**Q3**) In a conversation between two users, sending a message writes the conversation aggregate and the recipient's aggregate, whose count of unread messages goes up. How many writes does one message now make?

Each of the two aggregates has two copies, and every write must reach both copies: four writes.

(**Q4**) Conversation C3 is the chat of a live-streamed concert: 10 000 messages are sent in ten minutes, and node 0 is busy. Would adding nodes help?

Adding nodes does not help the messages. All the messages of C3 write the same aggregate, and whatever the number of nodes, C3 is stored whole on one node, with its copy on one other: all its messages still land on the same node. Sharding spreads the aggregates over the nodes, not the load on one aggregate. Reads of C3 can be spread over its two copies, but the writes cannot: the remedy is smaller aggregates, for instance the messages of C3 split over ten aggregates by sender, which spreads the writes over the nodes but makes showing the chat read ten aggregates.