Race Conditions - A Practical Example

Karsten Meier @ JugHH 2024

Click to edit the Subtitle

About Myself

Principal Software Development Lead

Cubic Transportation Systems

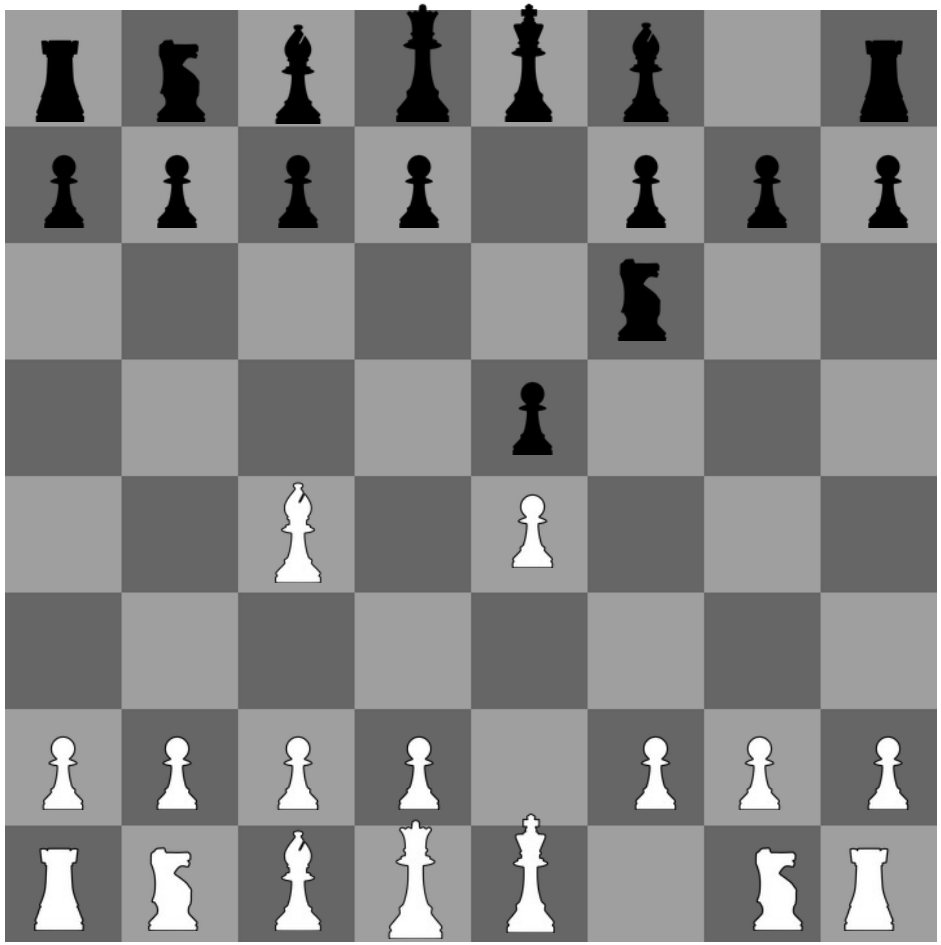
Till 30.4.24:

Senior Software Engineer

Hamburg Süd / Maersk

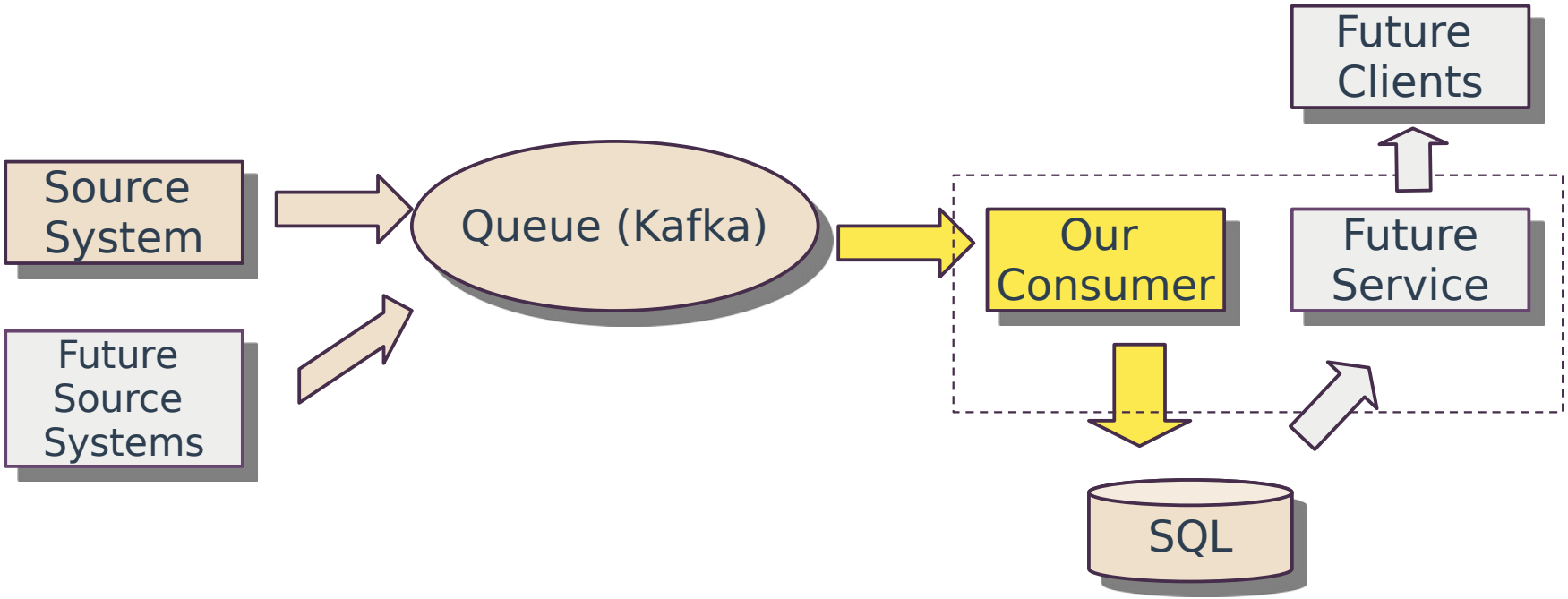
Aachen Limax, Magicman0361, CC BY-SA 4.0 via Wikimedia Commons

How Can White Win Fast?



Whenever we call an external system, the database, an API, it is one others turn, and our wonderful plan can be destroyed!

We Have a Simple Setup with High Throughput



We Consume Records and Map External to Internal Ids

External ID	Internal ID	(Version)	SomeData
LGCY-A	NEW-1	1	Chennai – Rotterdam
LGCY-B	NEW-2	1	Mumbai - Athens
LGCY-C	NEW-3	1	Cape Town - Miami
LGCY-A	NEW-1	2	Chennai - Hamburg

The Algorithm

```
var record = input.recieve()

var externalId = record.getExternalId()
var oldRecord =
   db.getLatestRecordWithExternalID( externalID )

if (oldRecord == null) {
   record.setInternalId( generateId() )
} else {
   record.setInternalId( oldRecord.getInternalId() )
}

db.save(record)
```

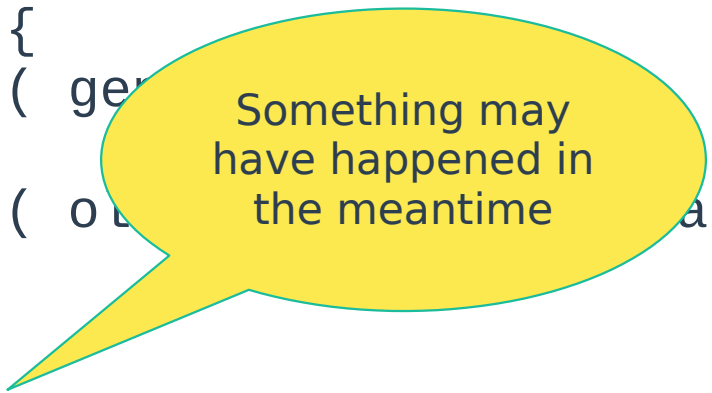
The Problem

```
var record = input.recieve()

var externalId = record.getExternalId()
var oldRecord =
    db.getLatestRecordWithExternalID( externalID )

if (oldRecord == null) {
    record.setInternalId( ge
} else {
    record.setInternalId( ol                alId() )
}

db.save(record)
```



Source system sends sometimes several versions instantly

External ID	SomeData
LGCY-E	Something
LGCY-E	Something’

Knowing the Problem is the First Step to Find a Solution

Photo: Karsten Meier, CC BY-SA 4.0

How can we solve
this special
race condition?

We Came Up With These Possible Solutions

Solution Idea
Custom Query to insert only when not exist
Transaction with higher isolation level
Explicit DB Locks
Change table structure, use key constraints
App remembers all ids in progress, e.g. in hash or set
App maintains buffer and stores not immedeatly
Eventual consitency: something checks for problem and merge the versions

Idea: Combine Query & Insert in One SQL Statement

```
INSERT INTO our_table (externalId,...)
  SELECT {#record.externalId}, {#record.otherField},...
  WHERE NOT EXISTS
    (SELECT externalId FROM our_table
     WHERE externalId = {#record.externalId}
RETURNING *
```

Idea: can we utilize Transactions?

Transactional SQL database conforms to ACID Standard

- Atomicity
- Consistency
- Isolation
- Durability

Using Transactions can often make the db more performant

SQL Defines Different Isolation Levels

```
BEGIN TRANSACTION
    LEVEL READ UNCOMMITED
BEGIN TRANSACTION
    LEVEL READ COMMITED
BEGIN TRANSACTION
    LEVEL REPEATABLE READ
BEGIN TRANSACTION
    LEVEL SERIALIZABLE

SELECT ..
INSERT ..

COMMIT
```

- **Internally the database uses locks and snapshots**
- **Postgres “Read Commited” is standard**
- **Higher isolation levels may have a performance impact**
- **Works good with updating rows, but still the problem when the non-existence of data is a condition.**

Theoretically, Serializable should Work

13.2. Transaction Isolation

13.2.3. Serializable Isolation Level

The *Serializable* isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions had been executed one after another, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must be prepared to retry transactions due to serialization failures. In fact, this isolation level works exactly the same as Repeatable Read except that it also monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a *serialization anomaly* will trigger a *serialization failure*.

Idea: Use Explicit Locks

Locks in the database can be on tables, rows or pages.

But as we ask for the non-existence:

• row locking not working,

• table locking would block everything.

But Postgres has also Locks to use for application-defined meanings, called Advisory Locks.

```
SELECT pg_advisory_lock(1234567);

SELECT * FROM our_table WHERE ....

INSERT INTO our_table VALUES(....)

SELECT pg_advisory_unlock(1234567);
```

Idea: Refactor and Use a Key Constraint

External ID	Internal ID	Version	Origin	Destination
LGCY-A	NEW-1	1	Chennai	Rotterdam
LGCY-B	NEW-2	1	Mumbai	Athens
LGCY-C	NEW-3	1	Cape Town	Miami
LGCY-A	NEW-1	2	Chennai	Hamburg

External ID	Internal ID
LGCY-A	NEW-1
LGCY-B	NEW-2
LGCY-C	NEW-3

ID	Version	Origin	Destination
NEW-1	1	Chennai	Rotterdam
NEW-2	1	Mumbai	Athens
NEW-3	1	Cape Town	Miami
NEW-1	2	Chennai	Hamburg

The New Algorithm

```
var key = keyMapRepository.findByExternalId(externalId);
if (key == null) {
    try {
        var keyMapToSave = new KeyMap(externalId, generateId());
        keyMapRepository.save( keyMapToSave );
    } catch (DuplicateKeyException e) {
        key = findByExternalId( externalId );
    }
}
if (key != null) {
    versionToSave.internalId = key.internalId;
    dataRepository.save(versionToSave)
}
```

We can also handle the race condition in the consuming app.

Because Kafka can ensure
that all events with the same external ID
are handled by the same consumer

Rangierbahnhof Kornwestheim – Luftaufnahme
Rbfkornwestheim.jpg:, CC BY-SA 3.0,
via Wikimedia Commons

Strategy

When the handling of a record
with same external Id is in progress,

block until other handling has finished.

Maybe We Can Use Java “synchronized”?

```
var record = input.recieve();
synchronized( record.getExternalId() ) {
    doNormalSave( record );
}


public void doNormalSave (Record record) {
    var externalId = record.getExternalId();
    var oldRecord =
          db.getLatestRecordWithExternalID( externalID );
    if (oldRecord == null) {
        record.setInternalId( generateId() );
    } else {
        record.setInternalId( oldRecord.getInternalId() );
    }
    db.save(record);
}
```

Idea: Utilize “ConcurrentMap”

Concurrent Collections

The `java.util.concurrent` package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

(From the Oracle Java Tutorial)

First Try using “ConcurrentMap”

```
var idsInUse = new ConcurrentHashMap()
//...
idsInUse.putIfAbsent(externalId, "in progress")
doNormalSave(record)
IdsInUse.remove(externalId)
```

Idea: ConcurrentMap with Locks as Values

```
var idsInUse = new ConcurrentMap()
// ...
var lock = idsInUse.putIfAbsent(externalId, new Lock())
lock.lock()
doNormalSave()
lock.unlock()
idsInUse.remove(externalId)
```

Another Idea:

When the handling of a record with same external Id is in progress,

just get the internal id from the other handler

and use it without blocking.

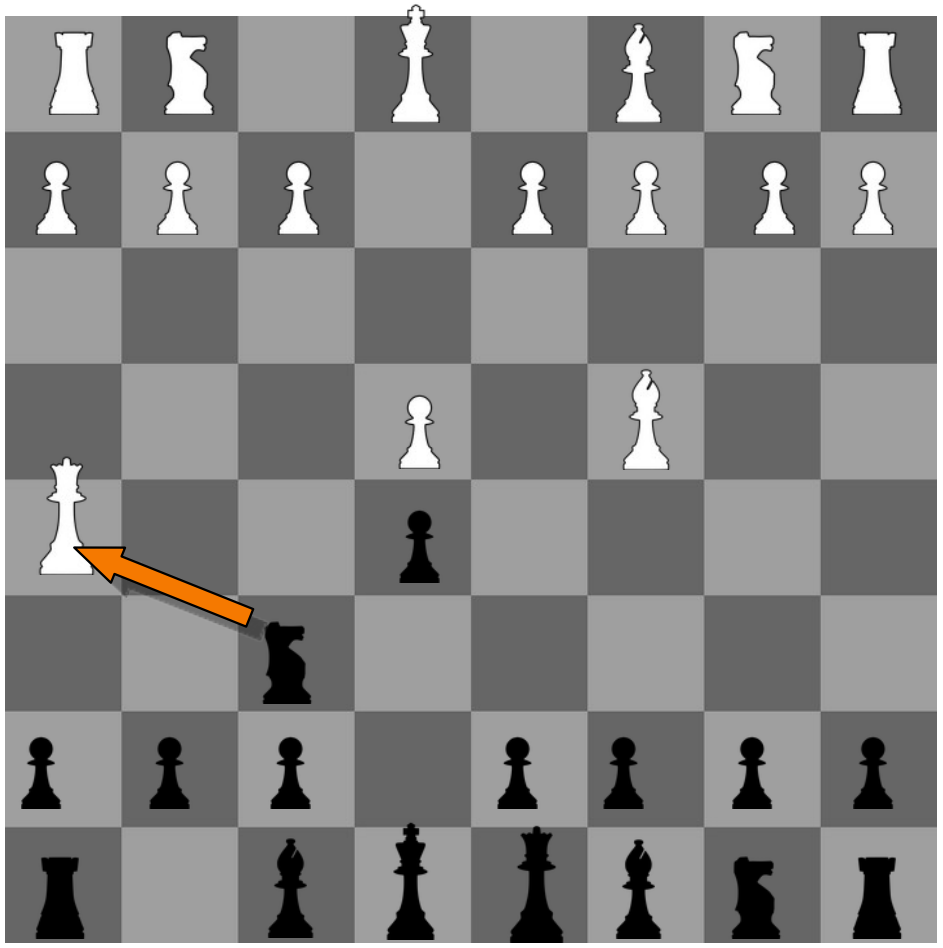
Code: Use the Id of the Saving in Progress

```
var oldRecord = db.getLatestRecordWithExternalID( externalID );
if (oldRecord == null) {
    var idCandidate = generateId();
    var internalId = idsInUse.putIfAbsent(externalId, idCandidate);
    record.setInternId( internalId );
    db.save(record);
    idsInUse.remove(externalID);
} else {
    record.setInternId( oldRecord.getInternalId() );
    db.save(record);
}
```

For our Use Case, Table Refactoring Seems to be Right

Solution Idea	Drawbacks
Custom Query to insert only when not exist	Still race condition possible inside the db itself
Transaction with higher isolation level	Seems to still need an explicit lock
Explicit DB Locks	Extra DB calls, also reactive transaction needed
Change table structure, use key constraints	Refactoring needed, but the new structure is a better fit to our use case
App remembers all ids in progress, e.g. in hash or set	Fears that it will be unreliable
App maintains buffer and stores not immedeatly	Decreased latency, also not easy to make it correct
Eventual consistency: something checks for problem and merges the versions	Problem that events may already be published when rolling back

Always Think Like You Have the Black Pieces



So this was our experience with a race condition.

https://meier-online.com