Distributed Sagas

A Protocol for Coordinating Microservices
Caitie McCaffrey
Distributed Systems Engineer

@caitie
caitiem.com
Monoliths
Microservices & NoSQL
Feral Concurrency Control:
An Empirical Investigation of Modern Application Integrity

Peter Bailis, Alan Fekete†, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

ABSTRACT

The rise of data-intensive “Web 2.0” Internet services has led to a range of popular new programming frameworks that collectively embody the latest incarnation of the vision of Object-Relational Mapping (ORM) systems, albeit at unprecedented scale. In this work, we empirically investigate modern ORM-backed applications’ use and disuse of database concurrency control mechanisms. Specifically, we focus our study on the common use of feral, or application-level, mechanisms for maintaining database integrity, which, across a range of ORM systems, often take the form of declarative correctness criteria, or invariants. We quantitatively analyze the use of these mechanisms in a range of open source applications written using the Ruby on Rails ORM and find that feral invariants are the most popular means of ensuring integrity (and, by usage, are over 37 times more popular than transactions). We evaluate which of these feral invariants actually ensure integrity (by usage, up to 86.9%) and which—due to concurrency errors and lack of database support—may lead to data corruption (the remainder), which we experimentally quantify. In light of these findings, we present recommendations for database system designers for better supporting Rails is interesting for at least two reasons. First, it continues to be a popular means of developing responsive web application front-end and business logic, with an active open source community and user base. Rails recently celebrated its tenth anniversary and enjoys considerable commercial interest, both in terms of deployment and the availability of hosted “cloud” environments such as Heroku. Thus, Rails programmers represent a large class of consumers of database technology. Second, and perhaps more importantly, Rails is “opinionated software” [41]. That is, Rails embodies the strong personal convictions of its developer community, and, in particular, David Heinemeier Hansson (known as DHH), its creator. Rails is particularly opinionated towards the database systems that it tasks with data storage. To quote DHH:

“I don’t want my database to be clever! … I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you’ll have to pry that logic from my dead, cold object-oriented hands … I want a single layer of cleverness: My domain model.” [55]
Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity

Peter Bailis, Alan Fekete†, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

“Application-level Mechanisms for maintaining database integrity”
Reserve a Hotel

Diagram showing the process of reserving a hotel:

1. User accesses the hotel booking website on a mobile device.
2. The Front End of the website communicates with the database.
3. The database retrieves information about available hotels and room types.
4. The website displays the available options to the user.
5. The user selects a hotel and room type and proceeds to payment.
6. The transaction is processed through a payment gateway, and the user receives a confirmation.
Reserve a Hotel

Feral Concurrency Control Mechanisms
Reserve a Hotel

Feral Concurrency Control Mechanisms
Reserve a Car
Trips

Front End

Database

Hotel Booking

Car Rental

Flight Booking

Payment Gateway

Database

Database
Trips

Front End

Feral Concurrency Control Mechanisms
Trips

Feral Concurrency Control Mechanisms
Death Star Architectures
Can We Do Better than Feral Concurrency Control?
Spanner: Google’s Globally-Distributed Database


Google, Inc.

Abstract

Spanner is Google’s scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner. Consistency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner’s main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [30].) Many applications at Google have chosen to use Megastore [5] because of its semirelational data model and support for synchronous repli-
Spanner: Google’s Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh

“Spanner is Google’s scalable, multi-version, globally distributed, and synchronously-replicated database”
Challenges to Adopting Stronger Consistency at Scale

Phillipe Ajoux*, Nathan Bronson*, Sanjeev Kumar*, Wyatt Lloyd†*, Kaushik Veeraraghavan*
*Facebook, †University of Southern California

Abstract

There have been many recent advances in distributed systems that provide stronger semantics for geo-replicated data stores like those underlying Facebook. These research systems provide a range of consistency models and transactional abilities while demonstrating good performance and scalability on experimental workloads. At Facebook we are excited by these lines of research, but fundamental and operational challenges currently make it infeasible to incorporate these advances into deployed systems. This paper describes some of these challenges with the hope that future advances will address them.

1 Introduction

Facebook is a social network that connects 1.35 billion people [19]. Facebook’s social graph reflects its users, their relationships, the content they create, and the actions they take. The social graph is large and constantly growing and changing. Data from this graph is stored in several geo-replicated data stores and tightly integrated with other services like Search and News Feed. No single data placement strategy can efficiently serve all workloads in a heavily sharded system, so many of these services choose a specialized sharding function and maintain their own data store, caches, and indexes.

The biggest barrier to providing stronger consistency guarantees in an environment like Facebook’s is that the consistency mechanism must integrate consistency across many stateful services. All of the scaling mechanisms described above lead to extra copies of data. Even if each of our caches and independent services were linearizable [24], inconsistencies would still be present in the aggregated result. Inter-service tracking is complicated by services that store data derived from a query to a lower layer, making object-level tracking insufficient. While these challenges are acute for Facebook at its present scale, we first encountered them when Facebook was much smaller. Solutions to these problems will benefit the growing class of applications whose implementation relies on sharding and separation into stateful services.

Another fundamental challenge is that a general consistency mechanism at scale must tolerate high query rates with low latency. Popular systems like Facebook have seen a recent spike in query volume, with results that sometimes take many seconds to complete. When we use APIs to interact with Facebook, the system returns a “Wait” response to our requests, preventing our application from moving forward while the request is processed. This delay is very damaging to customer experience, and so finding solutions to these consistency challenges is critical for Facebook’s long-term success.
Challenges to Adopting Stronger Consistency at Scale

Phillipe Ajoux*, Nathan Bronson*, Sanjeev Kumar*, Wyatt Lloyd†*, Kaushik Veeraraghavan*

*Facebook, †University of Southern California

“...is that consistency mechanisms must integrate across many stateful services”
Two Phase Commit
2PC: Prepare

Propose Book Hotel

Propose Book Car

Propose Book Flight
2PC: Prepare

- Front End
- Vote Book Hotel
- Vote Book Car
- Trips
- Vote Book Flight
2PC: Commit

Commit/Abort Book Hotel
Commit/Abort Book Car
Commit/Abort Book Flight
2PC: Doesn't Scale

- \( O(N^2) \) Messages in the worst case
- Coordinator is a Single Point of Failure
- Reduced Throughput
Distributed Sagas

A Protocol for Coordinating Microservices
SAGAS

Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N J 08544

Abstract

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions. To alleviate these problems we propose the notion of a saga. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution. Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly. We analyze the various implementation issues related to sargas, including how they can be run on an existing system that does not directly support them. We also discuss techniques to exploit the speedup provided by sargas for transaction processing.

The majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a].

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a]. To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction. As a consequence, other transactions wishing to access the LLT's objects suffer a significant delay.
“Sagas are Long Lived Transactions [in a single relational Database]”
“A Saga is a Long Lived Transaction that can be written as a sequence of transactions that can be interleaved.

All transactions in the sequence complete successfully or compensating transactions are ran to amend a partial execution.”
Distributed Sagas

A Protocol for Coordinating Microservices
A Distributed Saga is a Collection of Requests

- Book Hotel
- Book Car
- Book Flight
- Charge Money
A Distributed Saga is a Collection of Requests

Book Hotel  Book Car  Book Flight

Charge Money

and Compensating Requests

Cancel Hotel  Cancel Car  Cancel Flight

Refund Money
A Distributed Saga is a Collection of Requests

- Book Hotel
- Book Car
- Book Flight
- Charge Money

and Compensating Requests

- Cancel Hotel
- Cancel Car
- Cancel Flight
- Refund Money

that represent a single business level action
Distributed Saga Requests
Requests
Can Abort

Book Car

{ "Name": "Caitie McCaffrey", "Destination": "Malaga, Spain", "Start Date": "2017-05-17", "End Date": "2017-05-20" }
Requests

Can Abort

Book Car

{ "Success": false,
  "Message": "No Cars Available"
}
Requests
Must Be Idempotent

```
{
    "Name": "Caitie McCaffrey",
    "Destination": "Malaga, Spain",
    "Start Date": "2017-05-17",
    "End Date": "2017-05-20"
}
```
Requests
Must Be Idempotent

{  
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20"
}
Requests

Must Be Idempotent

Book Car

```json
{
    "Success": "true",
    "Confirmation Number": "ABC456"
}
```
Requests

Must Be Idempotent

```json
{
    "Name": "Caitie McCaffrey",
    "Destination": "Malaga, Spain",
    "Start Date": "2017-05-17",
    "End Date": "2017-05-20"
}
```

Book Car

```json
{
    "Success": "true",
    "Confirmation Number": "ABC456"
}
```
Requests
Must Be Idempotent

```
{
    "Name": "Caitie McCaffrey",
    "Destination": "Malaga, Spain",
    "Start Date": "2017-05-17",
    "End Date": "2017-05-20"
}
```
Requests

Must Be Idempotent

Book Car

Timeout
Requests

Must Be Idempotent

Book Car

{ 
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20"
}
Requests
Must Be Idempotent

Book Car

```json
{
    "Success": "true",
    "Confirmation Number": "ABC456"
}
```
Distributed Saga
Compensating Requests
Compensating Requests
Semantically undoes the effect of a request
Compensating Requests

Can Not Abort

{ "Success": true,
  "Confirmation Number": "ABC456"
}
Compensating Requests Must Be Idempotent
Compensating Requests

Must Be Commutative with Requests

Book Car

is the same as

Cancel Car

Book Car

Cancel Car
Compensating Requests
Must Be Commutative with Requests

Book Car
Compensating Requests
Must Be Commutative with Requests

Book Car
Timeout
Compensating Requests

Must Be Commutative with Requests
Compensating Requests
Must Be Commutative with Requests

- Book Car
- Cancel Car
Compensating Requests

Must Be Commutative with Requests

Book Car

Cancel Car
Requests

Idempotent
Can Abort

Compensating Requests

Idempotent
Commutative
Can Not Abort
Distributed Saga Guarantee

All requests were completed successfully
Distributed Saga Guarantee

All requests were completed successfully

Book Hotel   Book Car   Book Flight   Charge Money

Or a subset of requests and the corresponding compensating requests were executed

Book Hotel   Book Car   Cancel Hotel   Cancel Car
Distributed Saga Guarantee

No Atomicity

No Isolation
Distributed Saga Guarantee

No Atomicity

No Isolation

Visible before Saga Completes

Book Hotel

Book Car
Distributed Saga Guarantee

All requests were completed successfully:
- Book Hotel
- Book Car
- Book Flight
- Charge Money

Or a subset of requests and the corresponding compensating requests were executed:
- Book Hotel
- Book Car
- Cancel Hotel
- Cancel Car
defining a Distributed Saga
Directed Acyclic Graph

Distributed Saga

Start Saga

Car

Hotel

Payment

Flight

End Saga
Distributed Saga Vertex

Name: Hotel
Request: Book Hotel
Compensating Request: Cancel Hotel
Status: Not Completed
Distributed Saga
Start & End Vertices
Distributed Saga Log

fault-tolerant & highly available
Saga Execution Coordinator

Saga Log

Start Saga
- Car
- Hotel
- Flight
- Payment

End Saga
Executing a Distributed Saga
Book Trip Request

```json
{
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20",
  "Payment Token": "Tm90IG15IjJyWwgY3J1ZWgl0IGNhcncmQgaW5mbyA6KQ==",
  "Price": "2500USD"
}
```
Book Trip Request

```json
{
    "Name": "Caitie McCaffrey",
    "Destination": "Malaga, Spain",
    "Start Date": "2017-05-17",
    "End Date": "2017-05-20",
    "Payment Token": "Tm90IG15IHJlYWwgY3J1ZGl0IGNhcmQgaW5mbyA6KQ==",
    "Price": "2500USD"
}
```
Book Trip Request

```json
{
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20",
  "Payment Token": "Tm90IG15IHJlYWwgY3J1ZGl0IGNhcmQgaW5mbyA6KQ==",
  "Price": "2500USD"
}
```
Book Hotel Request

```json
{
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20"
}
```
Car
Flight
Hotel
Payment

Start Saga
End Saga

Saga Log

Book Hotel
Response

{ "Success": "true",
"Confirmation Number": "WXY123"
}

Start Saga

Car
Hotel
Flight

Payment
End Saga
Start Saga
Start Hotel
End Hotel
Start Car

Saga Log

Book Car Request

```
{
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20"
}
```

Done

End Saga

Car

Hotel

Flight

Payment
Start Saga

Start Hotel

End Hotel

Start Car

End Car

Start Saga

Hotel

Flight

Payment

End Saga

Saga Log

Book Car Response

{ "Success": "true", "Confirmation Number": "ABC456" }
 Saga Log

Start Saga

Start Hotel

End Hotel

Start Car

End Car

Start Flight

Book Flight Request

```json
{
  "Name": "Caitie McCaffrey",
  "Destination": "Malaga, Spain",
  "Start Date": "2017-05-17",
  "End Date": "2017-05-20"
}
```

Done

Book Flight Request

Payment

End Saga

Start Saga

End Saga
Book Flight Response

```
{
  "Success": "true",
  "Confirmation Number": "789QPZ"
}
```
Start Saga

End Saga

Start Hotel

End Hotel

Start Car

End Car

Start Flight

End Flight

Payment

Start Saga

Car

Hotel

Flight

End Saga
Start Saga
Start Hotel
End Hotel
Start Car
End Car
Start Flight
End Flight
Start Payment

Saga Log

Payment Request
{
  "Payment Token": "Tm90IG15IHJ1YWwgY3JlZGl0IGNhcmQgaW5mbmlyb3I=",
  "Amount": "2500USD"
}

End Saga
Failure of a Distributed Saga
Start Saga

Book Car Request

```
{
    "Name": "Caitie McCaffrey",
    "Destination": "Malaga, Spain",
    "Start Date": "2017-05-17",
    "End Date": "2017-05-20"
}
```
Saga Log

- Start Saga
- Start Hotel
- End Hotel
- Start Flight
- Start Car
- Abort Car

```
{
  "success": "true",
  "Confirmation Number": "WXY123"
}
```

Start Comp Saga

End Comp Saga

Flow Diagram:
- Car
- Hotel
- Flight
- Payment

- Comp Hotel
- Cancel Hotel Response
- Start Comp Saga
{ "Success": "true", "Confirmation Number": "789QPZ" }
Start Saga

Start Hotel

End Hotel

Start Flight

Start Car

Abort Car

Comp Hotel

End Flight

Saga Log

Start Saga

Start Hotel

End Hotel

Start Flight

Start Car

Abort Car

Comp Hotel

End Flight

Start Comp Saga

End Comp Saga

Car

Hotel

Flight

Payment

Done

Cancel Flight Request

{ "Name": "Caitie McCaffrey", "Confirmation Number": "789QPZ" }
Saga Log

Start Saga

Start Hotel

End Hotel

Start Flight

Start Car

Abort Car

Comp Hotel

End Flight

Comp Flight

Payment

Start Comp Saga

End Comp Saga

End Saga

SEC

Car

Hotel

Flight

Payment

Start Comp Saga

End Comp Saga
Distributed Saga Guarantee

All requests were completed successfully

Or a subset of requests and the corresponding compensating requests were executed
Recovering from Saga Execution Coordinator Failure
Isolation of Complex Code
with Distributed Sagas

Complex Code Lives Here

Complex Code Lives Everywhere
Modular Services
with Distributed Sagas

Only Flight Booking

Flight Booking & Feral Concurrency Control Mechanisms
Service Composition with Distributed Sagas

Just Add A New Saga!

New Service & Feral Concurrency Control Mechanisms
Distributed Sagas Makes Building & Modifying Microservices Easier
Thank you

@caitie