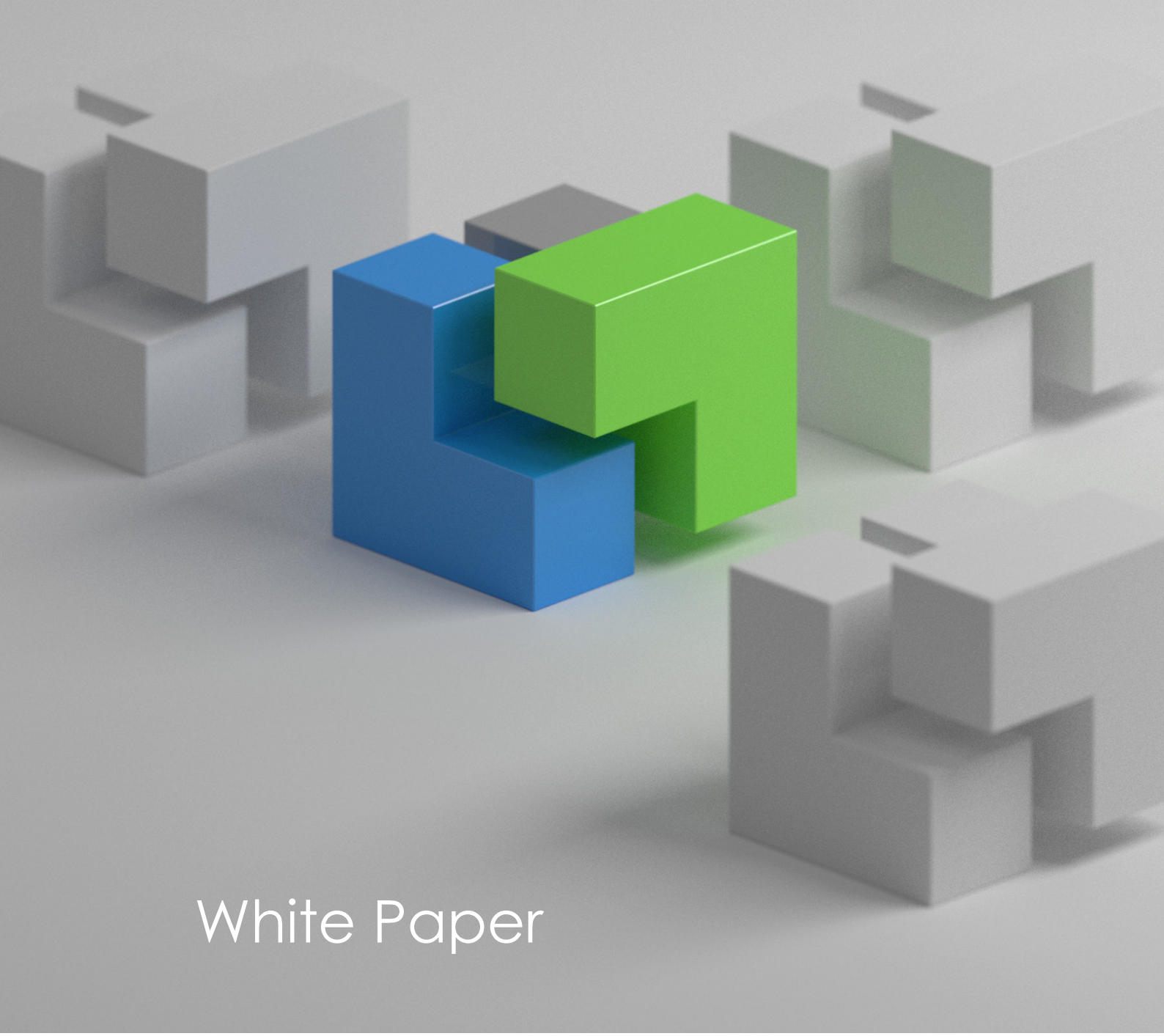

White Paper

Migrating to MongoDB

New thinking in working with databases

Bricknode Financial Systems

Bricknode Financial Systems (BFS) is a cloud based platform for financial services firms. BFS is comprised by a number of web applications for e.g. back office, investment brokers, asset managers, investment advisors and end customers. These front ends are serviced by a common backend which is comprised by a number of business logic modules, some core modules including order and transaction handling, adapter modules for integration with our customers' partners (custodians, insurance companies etc.) and a data layer for business object persistence.

Business objects

For business objects in BFS, e.g. orders, accounts, instruments etc., we use an abstraction called "Brick". The Brick contains functionality common to all business objects mostly regarding getting or setting attributes and serialisation for sending it over the Internet or storing it in a database.

"After migrating to MongoDB our complete data export time went from over 7 minutes to 53 seconds" - Erik Ekblom, Senior developer at Bricknode

Data layer

From start we chose PostgreSQL, an advanced open source SQL database, for data storage. To support flexibility and make the system easy to extend, we wanted to make the database "schema less", i.e. the attributes of a business object (e.g. for an order: order number, business date, settlement date) are dynamic. This means that one order could have a different set of attributes from another order. Custom attributes could be added to Bricks as well. The implementation chosen to facilitate this flexibility is an Entity-attribute-value model (EAV-model). Although providing incredible flexibility the EAV-model has its drawbacks. For instance querying could be a complex task compared to traditional relational modeling and therefore we implemented our own

query objects for standard queries as well as ad-hoc queries. The data layer has functionality for transforming a BFS-query to a set of SQL-statements which return the requested data.

Other functionality provided by the data layer include a complete audit trail of business object manipulation implemented by database triggers and atomic transactions for transfer of financial assets.

Motivation

When performing complex queries to large databases, the EAV-model tends to consume a lot of computing resources. The database server was becoming a bottleneck, since we could not find an easy-enough way to scale it out. This made us search for alternative data storages and the recent advent of "NoSQL" databases provided us with a few interesting candidates.

MongoDB

MongoDB (from "humongous") is an open source "NoSQL" database. Our reasons for choosing MongoDB were its document-oriented storage which features "schemalessness", replication functionality that provides high availability and "auto-sharding" which provides completely horizontal scalability. Furthermore our initial tests were very promising regarding performance for querying as well as updates.

The great performance of MongoDB comes with the cost of some lacking functionality. Table joins are not possible, there is no functionality to enforce referential integrity and no support for atomic transactions over multiple objects.

Challenges

Coupling

When we started to develop the data layer on top of PostgreSQL, we were aware of the possibility that we would switch the database engine in the future. Therefore we were strict in our layered architecture and kept coupling between the layers to a minimum. We defined an interface for the data layer and developed an implementation of that interface for PostgreSQL "according to the textbook". It was then easy to make the choice of data layer implementation a configuration matter.

Missing features

The advantages of MongoDB, primarily its performance, scalability and replication features, outweighs the missing functionality and we decided to implement the features we needed in BFS. Since we have implemented some fundamental database functionality in the data layer, it is important that we are disciplined and make all access to the database through the data layer only. One great advantage of this approach is the possibility of multiple storage engines and using the one with the right properties for a specific part of the system.

Referential integrity is ensured by the data layer by querying the database before inserts, updates and deletes. This has some impact on database writes but none on reads.

At the moment, the part of BFS that is dealing with financial transactions is still based on PostgreSQL. We are currently developing a new transaction module which will use MongoDB and it will probably ensure atomic transactions by implementing "two phase commit" in the data layer.

Because of the EAV-model used in our PostgreSQL data layer and our in-house developed query functionality described earlier, the implementation of table joins was dealt with by developing the query functionality for MongoDB as well.

There is no support in the current version of MongoDB for fixed precision numbers. BFS depends heavily on fixed-point arithmetic, since it is a financial platform. The solution was to make conversions (by using simple multiplication and division) in the data layer and store the numbers as integers.

Data migration

With the architecture described above, it didn't take very much effort to develop a simple program that could connect to two data layers, read all the Bricks from one and save them to the other.

Conclusion

We are really satisfied with the performance of MongoDB and our users as well. Perhaps we had higher expectations on some features, e.g. MapReduce, but on the other hand the aggregation framework exceeds them. Its scalability and replication functionality suit our needs very well as do the interfaces to it, drivers as well as the environments for ad-hoc access. We do look forward to some features that seem to be in the development pipeline of MongoDB, e.g. asynchronous I/O in the driver and a fixed precision data type.

By following healthy development practice, mostly regarding modularisation (principles of coupling and cohesion), transition between database engines can be quite seamless.

Author

Erik Ekblom, Senior Developer and co-founder at Bricknode
E-mail: erik@bricknode.com

Erik works as a developer at Bricknode focusing on core framework and database technology. Erik is also responsible for various asset management products.

Who are Bricknode?

Bricknode is a young innovative company focusing on financial software. Our mission is to empower our business partners to deliver the best financial products to their end customers and partners in the most efficient way possible.

Bricknode Financial Systems is a cloud-based platform, which makes it possible for securities firms and other financial institutions to establish themselves online.

The costs of operating a financial institution are increasing in lockstep with tighter regulation and compliance requirements. By utilizing the latest technology within cloud-based services Bricknode can enable financial companies to share services and initiate partnerships. This all funnels out to being able to offer great products to the end user.

BRICKNODE

Bricknode develops and maintains a complete cloud-based securities system based on the concept Software as a Service (SaaS).

Using Bricknode Financial Systems security companies can offer complete online banking through relatively small investments and costs. Read more about Bricknode and Bricknode Financial Systems at www.bricknode.com.

Phone: +46 (0)8-559 22 180
Reg.nr: 556780-7564

Bricknode Stockholm

Hamngatan 11
S-111 47 Stockholm
Sweden

Bricknode Skövde

Visiting address:

Kanikegränd 3B
S-541 34 Skövde
Sweden

Address:

Box 133
S-541 23 Skövde
Sweden