Master’s Thesis Nr. 133
Systems Group, Department of Computer Science, ETH Zurich

Decoupling of Data Processing Systems over RDMA/InfiniBand

by
Alessandro Dovis

Supervised by
Prof. Dr. Gustavo Alonso
Claude Barthels
Jana Giceva
Darko Makreshanski

February 2015–August 2015
Abstract

Modern workloads are becoming more and more diverse and involve a large volumes of frequently updated data. On such workloads, enterprises often have strict requirements with regards to data freshness and performance guarantees. This demand can be fulfilled in a scalable way by engineering data processing systems that are inherently distributed and rack-scale computers can be perfect architectural platforms for the deployment of such systems. Rack-scale computers are becoming the building blocks of modern data centers; they are characterised by new hardware technologies that promise terabytes of main memory and thousands of CPU cores connected by low-latency, high-bandwidth interconnects, such as InfiniBand or silicon photonics.

FBX is a distributed data storage and processing engine, designed to support different types of workload, from transactional processing, to analytics, to machine learning and graph processing. Contrary to the widespread trend of creating separate specialised systems for each type of workload, the design of FBX tries to unify them into a single solution, by having a single storage engine that exposes a common interface to the data processing layers; the storage engine can be distributed, replicated and employ different specialised backends, but its implementation details are hidden from the data processing engines.

This thesis presents how SharedDB, FBX’s single-machine parent, has been decoupled and distributed, exploiting RDMA/InfiniBand interconnect and the Message Passing Interface (MPI) library. The design and implementation are discussed in depth and the system performance is evaluated with a variety of standard benchmarks.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Context and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Research Questions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>2</td>
</tr>
<tr>
<td><strong>2 Background and Related Works</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 RDMA/InfiniBand</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Message Passing Interface (MPI)</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Data storage/processing systems using RDMA</td>
<td>10</td>
</tr>
<tr>
<td>2.4 SharedDB and Crescando</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 System Distribution Design</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1 SharedDB Query Plan</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Principles and Design Space</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Stub Operators Detailed Design and Implementation</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1 Networking Layer API</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Result Structure: Translation and Compaction</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3 Message Reordering</td>
<td>22</td>
</tr>
<tr>
<td>3.3.4 Optimisations</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Multiple Query Engine Design</td>
<td>24</td>
</tr>
<tr>
<td>3.4.1 Result Multicasting and Filtering</td>
<td>24</td>
</tr>
<tr>
<td>3.4.2 Query ID Space Management</td>
<td>25</td>
</tr>
<tr>
<td>3.5 Current and Future Work</td>
<td>26</td>
</tr>
<tr>
<td><strong>4 Networking Layer Implementation</strong></td>
<td>29</td>
</tr>
<tr>
<td>4.1 Choice of MVAPICH MPI</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1 Basic Benchmarks</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Big Messages Management</td>
<td>32</td>
</tr>
</tbody>
</table>


**Contents**

4.2.1 Issues with Point-to-Point Semantics 33
4.3 Small Messages Management 34
4.4 Hybrid Solution 36
4.5 Multicast Experiments 37
4.6 Networking Layer Resource Management and Message Prioritisation 38

5 Evaluation 41
5.1 Description of Benchmarks and Experimental Setup 41
5.2 General Evaluation 43
5.2.1 Analytical Workloads 43
5.2.2 Transactional Workloads 46
5.2.3 Mixed Workloads 47
5.3 Network Overhead Analysis 47
5.3.1 Analytical Workloads 47
5.3.2 Transactional Workloads 51
5.4 Further Benchmarks of Specific Features 52
5.4.1 Resource Management 52
5.4.2 Message Prioritisation 53

6 Conclusion 55
6.1 Contributions 55
6.2 Future Work 56

Bibliography 57
1.1 Context and Motivation

The world of databases and data storage/processing systems is experiencing an incredibly fast-paced evolution: the diversity of workloads is increasing and new exotic applications – such as machine learning and graph processing – are pulled alongside more traditional ones, such as transactional and analytical processing; furthermore, the demand for performance guarantees, high speed and data freshness are constantly escalating. This undermines standard database systems, that need to transform, in order to keep up with the new requirements. Most commercial solutions have moved towards specialisation of multiple systems for different workloads: each of this systems will perform very well on a few specific tasks, but the maintenance and integration of multiple systems becomes complex, in case different workloads need to access the same data (they will need to implement complex publish-subscribe mechanisms, as in Oracle GoldenGate [6], or expensive cross-engine data migrations, as in BigDAWG [11]).

FBX is a modular system designed to solve this problem, by supporting all the aforementioned workloads in the same system: a single storage engine contains all the data and exposes a common batch-based interface to the data processing engines (or query engines). Query engines are customisable modules that are tailored to a specific workload or application and can be implemented adopting query batching, query-at-a-time stored procedures or other data processing strategies. The storage engine can be internally distributed, with data replicated on different machines, employ different backends for queries with different performance profiles and provide freshness guarantees via transparent update propagation.

Our system should attain high performance in a scalable setting, since our solution is designed mostly for workloads that couldn’t fit in a single-machine system. For this reason, we embrace the rack-scale computer architecture as
1. Introduction

infrastructure to deploy our system on. Such a framework provides thousands of cores, terabytes of RAM and fast interconnects. In particular, its high-speed network matches perfectly our vision, as both high throughput and low latency are required from different workloads.

In the thesis, we describe the first steps in building FBX. The project started from a single-machine multicore database, designed around query and data batching and computation sharing. In order to become a distributed system, the storage engine had to be decoupled from the query engines and a data transfer protocol needed to be designed, that could satisfy the requirements of different workloads.

We present the design and implementation of these aspects of the system, together with the evaluation of its performance on multiple workloads. We omit the details about other parts of the system (storage engine implementation, replication strategy, data processing operators), since they are not a direct contribution of this thesis.

1.2 Research Questions

In particular, we want to answer the following questions:

• How to decouple storage engine operators from query engine (data processing) operators in a scalable and configurable way?

• How to transfer queries and data between operators on remote machines efficiently?

• How to deal with both latency-sensitive and throughput-sensitive workloads?

1.3 Outline

The next chapters are structured as follows:

• Chapter 2 first gives an introduction to RDMA, to the InfiniBand architecture and to the Message Passing Interface (MPI) library; next, it presents a few works in the field of data storage and processing systems that utilise RDMA and eventually it describes the main traits of the systems that lent most of its initial codebase to FBX;

• Chapter 3 focuses on the decoupling mechanisms design, starting from the principles and design options, moving to the details of the chosen design, some implementation aspects and optimisations; in the final sections, it explains the enhancements to the design in the presence of multiple query engines;
1.3. Outline

• Chapter 4 provides a detailed description of the networking layer, its algorithms to deal with different message types and sizes and its internal resource management;

• Chapter 5 evaluates the design and implementation described in the previous chapters, employing standard OLTP, OLAP and mixed workload benchmarks;

• Finally, Chapter 6 contains a summary of the main contributions and future work.
Chapter 2

Background and Related Works

This chapter presents the system we started from, the main technologies we used to build FBX and some projects that are similar to ours in terms of target or techniques. Only the main concepts are described, focusing on the details that matter in the design and implementation of our system; more extensive explanations can be found in the papers and web-pages referenced throughout the chapter.

First, we give an introduction to RDMA/InfiniBand, the network architecture that we employ in the project. Next, an overview of MPI – a standard communication protocol extensively used in the area of high performance computing – is provided, together with a few details about some of its implementations that have InfiniBand support. The following section presents some related projects in the field of data management and processing that make use of RDMA as inter-machine communication protocol. We conclude the chapter with a brief description of the systems that we employed as starting point for the development of our distributed engine.

2.1 RDMA/InfiniBand

A Remote Direct Memory Access (RDMA) protocol is a “wire protocol that supports RDMA Operations to transfer ULP data between a Local Peer and a Remote Peer.” [18]. RDMA provides several key features:

- **OS kernel bypass**: messages are sent between user-space processes without kernel intervention;
- **asynchronous communication interface**: operations are performed in a nonblocking fashion, allowing a high level of communication-computation overlap;
- **processor bypass**: the processor is not involved in the memory copy,
2. **Background and Related Works**

since DMA operations between memory and network adapters are performed;

- **explicit buffer management**: memory buffers need to be registered explicitly before any RDMA operation can be performed with them and deregistered after the operations complete.

Altogether, these characteristics result in low latency, high throughput and low CPU utilisation. On the other hand they burden the application with some additional complexity, as it needs to manage its own buffers in order to exploit the power of RDMA effectively.

To date, multiple implementations of RDMA have been developed, such as InfiniBand, RDMA over Converged Ethernet (RoCE) [9] and iWarp [17]; the first two implement *true* RDMA over different physical and link-layer protocols, while the third solution is a layer on top of standard TCP or SCTP.

*InfiniBand* (IB) [8] is a communication standard between processors and I/O devices. It defines the interconnection among devices using a high-speed switched network fabric topology; the nodes are connected to the fabric through channel adapters: processors use Host Channel Adapters (HCAs), while I/O controllers contain Target Channel Adapters (TCAs).

The asynchronous interface is implemented via the concept of queues: in a point-to-point communication, each host has a *Queue Pair* (QP), comprising a send and a receive queue, and a *Completion Queue* (CQ). When an application wants to send or receive a message, it posts a *Work Request* (WR) to the correct queue of the QP; once the operation is terminated, an event is posted to the CQ. While waiting for a completion event, the application can perform some computation, since the CPU is not involved in the communication. The application can discover new events either by polling/waiting on the CQ or by handling an interrupt (only by explicitly enabling interrupts and using *solicit* events). Both Receive Queues and Completion Queues can be shared among different connections: in the receive case, a Shared Receive Queue (SRQ) can be bound to multiple QPs at connection startup.

*Memory regions* (MRs) are registered with the HCA before they are used for RDMA operations: this step involves the OS and has the purpose of pinning the MRs to main memory, to permit DMA from the HCA. Once a MR is registered, any RDMA operation on it bypasses the kernel completely.

*InfiniBand* exposes multiple semantics for data transfer:

- **two-sided operations** (namely *RDMA Send* and *RDMA Receive*): these primitives involve both sides of the communication. The sender sends a message specifying a local buffer as source and the receiver posts a receive request specifying a local buffer as destination; once the transfer is completed, the receiver can find a completion event in the CQ. For the transfer to be successful, at least a receive request needs to be
performed strictly before the send request; if multiple receive requests are posted on the same Receive Queue, a buffer is picked arbitrarily from the buffers ready to receive.

- one-sided operations (namely RDMA Read and RDMA Write): these primitives allow to read/write memory directly to the remote side, without any remote CPU involvement. To make it possible, the passive side registers a buffer and communicates a capability – a "key" to access the remote buffer – to the active side; now the active side can read or write directly to that buffer by specifying a local buffer and the key of the remote buffer: the remote buffer is then transferred to the local buffer (or vice versa) without any notification to the passive side.

On the one hand, one-sided operations are the essence of RDMA and provide higher throughput; on the other hand, they can be harder to exploit as-they-are in a user application, since operations are unsignaled and – if signals are needed – a control mechanism must be implemented in an upper layer (in-band or out-of-band).

Finally, InfiniBand offers several Transport Modes. Reliable Connection (RC) binds each QP to a remote host QP; in Reliable Datagram (RD) each QP can send/receive messages to/from any remote QP; Unreliable Connection (UC) and Unreliable Datagram (UD) are analogous to the previous ones, but without acknowledgment signals and retransmission. Only RC, UC and UD are currently implemented; any of them can be used for message unicast; only UD can be used for message multicast.

Frey and Alonso [7] have analysed the costs of using RDMA in applications and have identified some optimisation strategies to follow in order to use RDMA efficiently:

- reuse buffers as much as possible, to minimise the number of MR registration/deregistration operations (which are costly and involve the OS);
- if this is not possible, memcpy application buffers into/from pre-registered MRs if the buffers to be sent/received are small, while register/deregister the buffers if they are big (the critical size depends on the relative performance of memory copy with respect to IB registration);
- overlap buffer management with communication, register pages already resident in main memory and parallelise the buffer registration.

These are all useful suggestions to keep in mind when designing the software layers directly on top of RDMA. When using a communication layer on top of RDMA, such as MPI, most of these strategies are already considered and implemented, but they should also drive the design of the upper layers, to make the best use of the library.
2.2 Message Passing Interface (MPI)

MPI [30] is the de-facto standard for inter-node communication in parallel applications in the area of high performance computing. The standard is transparent both to the machines running the application and to the networking technology being used, making it very portable; several implementations have been developed, both open-source and proprietary, with different support for a variety of computing and communication architectures.

The communication is based on the concept of **communicator**, a group of processes – potentially running on several machines – that can talk to each other by exchanging messages.

Various classes of operations have been defined:

- **point-to-point**: the communication happens between two nodes, which can send messages, probe for the arrival of a message and receive messages. The calls can be blocking or nonblocking: in the latter case, the control is given back to the application, which is responsible for checking for the completion by testing or waiting on the state of the operation;

- **collective**: the communication involves all the members of the group. For instance, this class includes broadcast, scatter, gather, barrier, reduce and all-to-all; also in this case, operations can be blocking or nonblocking;

- **one-sided**: added in the MPI-2 standard, they introduce the RMA semantics into MPI. These calls perform writes or reads on remote memory, separating data transfer from synchronisation and letting the application define memory windows that are used in the communication.

In the implementation of the channels between different parts of our system, we will employ both point-to-point and one-sided operations.

In point-to-point communication, the standard doesn’t define the internal protocols used for delivering a message from sender to receiver, nonetheless most implementations define two basic strategies:

- **eager protocol**: it is an asynchronous send and can complete before the receiver has posted a receive operation; it leverages the presence of a pool of buffers on the receiver side and sends the message on the wire without any handshake, assuming that there is enough space at the receiver;

- **rendezvous protocol**: the two parties perform a handshake in which the size of the message is communicated to the receiver, which can make enough space available for it; once the handshake is over, the communication can be performed without the help of MPI-internal buffers.
2.2. Message Passing Interface (MPI)

Usually, the eager protocol is used for small messages and the rendezvous protocol for large messages.

Multiple MPI implementations support RDMA, such as OpenMPI [2] and MVAPICH2 [1] and much work has been done to design protocols that exploit RDMA features in an efficient and scalable way.

In the area of point-to-point communication [16, 19, 21, 26], both eager and rendezvous protocols have been extensively optimised to minimise the latency and maximise the throughput in order to achieve the same speed as raw InfiniBand, both for small and big messages.

J. Liu et al. [21] compare different InfiniBand semantics in the context of the eager protocol and propose the following design: a unidirectional RDMA channel uses a ring of coupled buffers (both on the sender and on the receiver side) with a head and a tail pointer; the sender writes into its head buffer (copying the application buffer into the MPI-internal ring) and RDMA-Writes it into the corresponding buffer on the receiver side; the receiver polls in-band flags on the local head buffer to check the arrival of a new message and – once it has read the message and copied it into an application buffer – sends explicit updates to the tail pointer to the sender (which results in a flow control mechanism).

Such design is reproposed and enlarged in a different paper by J. Liu et al. [19], where some optimisations are discussed, such as piggybacking of pointer updates and pipelining for large messages. Moreover, two zero copy designs are proposed and compared for the rendezvous protocol: in the first one, the sender advertises the buffer to the receiver, which can perform an RDMA-Read operation into its local buffer and signal the end of the operation via a control packet; in the second one, the receiver replies to the sender with a “ready” packet and the data transfer is performed with an RDMA-Write from the sender. To decrease the overhead of MR registration, the usage of a registration cache is suggested.

S. Sur et al. [26] focus on the various rendezvous design alternatives: in addition to the aforementioned RDMA-Write and RDMA-Read-based designs, a novel RDMA-Read-based protocol is presented, that uses interrupts and solicit_events to increase the computation-communication overlap.

Finally, G. M. Shipman et al. [16] describe how the previous designs can be made more scalable by using InfiniBand SRQs and dynamic connections.

Collective operations have historically been built on top of point-to-point operations. For instance, a broadcast operation can be implemented by propagating the message over a tree structure (usually a binomial tree), in which node-to-node communication exploits the send/receive primitives described before.

This approach is attractive, because it makes the implementation of collective primitives completely architecture-independent; however, in the presence of RDMA, more HW support can be exploited to improve their perfor-
2. Background and Related Works

Several designs for a broadcast that uses HW multicast have been proposed and implemented [20, 27]; the main challenges are the technical differences between MPI broadcast and RDMA multicast:

- RDMA multicast works on UD (Unreliable Datagram), while MPI broadcast is reliable;
- UD doesn’t provide ordering guarantees;
- a message sent over UD can only be as big as the RDMA MTU, while MPI broadcast works with messages of any size.

Despite the effort spent on reducing the inherent cost of these dissimilarities, a non-negligible software overhead is introduced by MPI (ACK messages, retransmission, message copy and garbage collection, segmentation/reassembly), which makes the HW-based broadcasting appealing only in the presence of a very large number of nodes.

Finally, one-sided operations have been previously implemented on top of point-to-point operations as well. In most early implementations, the actual transfer was performed only once the target issued a synchronisation operation. This is necessary in MPI implementations over a networking medium that forces the target to actively participate in any communication.

MPI can exploit the RMA verbs of RDMA to perform the same operations in a much more efficient way [28], eliminating the costs of send/receive semantics (active receiver, message matching, handshakes). Once MPI windows are created (which can imply an RDMA registration of the corresponding MR), the MPI Get and Put operations are directly mapped to RDMA Read and Write verbs.

2.3 Data storage/processing systems using RDMA

The project presented in this thesis is not the first application of RDMA in the database field. Following is a brief description of the most influential works that employ InfiniBand or other RDMA technologies to build distributed data processing and storage systems.

The last few years have seen the proliferation of key-value stores that use RDMA in the communication with their clients. Pilaf [13] is a distributed in-memory key-value store, implemented using Cuckoo hashing. It uses RDMA for GET operations, by performing direct RDMA-Reads from the server memory; PUT requests are sent using RDMA-Send messages and are processed actively by the server; read-write races are resolved optimistically, via the design of self-verifying records. FaRM [10] is a general purpose main-memory distributed computing platform that exposes the memory of a cluster of machines as a shared address
space; on top of it, a key-value store has been implemented. It uses Hopscotch hashing and has a symmetric design (all servers are also clients): GETs are implemented as direct RDMA-Reads, while PUTs are sent via RDMA-Write into a buffer in the server, which polls the buffer and then processes the request. The polling technique resembles the one described in [21].

Herd [12] is a very recent key-value system. It is designed to minimise the number of round-trips and the paper analyses extensively the tradeoffs of different InfiniBand semantics and transport modes. It implements requests (both GETs and PUTs) as RDMA-Writes over UC to a predefined server MR; responses are sent over UD via RDMA-Send to the clients. Such a system has been proven to be superior to the previous ones at least by a factor of 2.

HyPer [33] is a hybrid OLTP/OLAP main-memory database based on virtual memory snapshots. It has been extended to allow distribution of portions of the query processing layer onto several machines [29]. Both TCP and RDMA are analysed as network options, with RDMA proven to be a better choice. The communication is performed using an exchange operator, which does the serialisation of the tuples and pushes them to the networking layer (and acts in the opposite way on the other side of the transfer); the networking part is constituted by a multiplexer (a set of dedicated worker threads), that communicates with the exchange operator via queues and sends/receives messages to/from the HCA; a message pool is used to avoid the expensive registration of new buffers on the serving path.

Tell [25] is a shared-data database, which decouples the in-memory storage engine from the query processors and embeds the transaction manager into the query processor, allowing scalability, fault-tolerance and elasticity and supporting ACID transactions and complex SQL queries. The paper focuses on storage data structures and concurrency control design; the details of how RDMA is exploited are not described in depth.

Vectorwise [22] is an analytical DBMS originated from the MonetDB column-store project. MPI has been used to distribute the database [32], using parallel exchange operators that follow a pull-based Volcano model.

The previous systems have similarities with the design of the distribution in FBX, described in the following chapters. The differences among all of them are due to distinct – sometimes opposite – performance requirements, level of physical/logical data independence, storage/query engine architecture and supported workloads.

2.4 SharedDB and Crescando

The FBX project has not started from a clean-slate codebase, but instead reuses extensively the codebase of the previous SharedDB project, starting
2. Background and Related Works

from a solid single-machine multicore DB solution as a basic foundation.

*SharedDB* [15] is a database system designed to meet SLAs in high load situations for complex and highly dynamic workloads. It is based on the concepts of query batching and computation sharing between several queries. Queries share a global execution plan; each operator is an always-on entity (implemented as a single thread or as a set of parallel worker threads, affined to specific CPUs). Operators communicate with neighbouring operators in the plan via queues; they work in batches: they first get all the incoming queries from the queue (as a batch), interpret them and send the queries down the operator graph; then they wait for the results concerning the current batch, process them and send the results to the consumers (neighbouring operators that sent the queries).

Such a push-based batching strategy allows work sharing among queries: supposing that two queries are interested in a join of two tables with different filtering predicates, SharedDB will perform the join of the union of the tuples that both queries are interested in, in both tables. This looks like a bad idea – and indeed it is, in low throughput situations – but its power is undeniable in high throughput situations, when work sharing can be performed effectively and operators are able to handle high load and provide response time guarantees.

Its data-query model consists of an intermediate representation of results as tuples, containing both the relation attributes and the IDs of the queries interested in the tuple. Implementation-wise, the query IDs can be carried in the form of arrays or bitsets (depending on the sparsity of the IDs). The system is logically tuple-based, but results are batched, for performance reasons.

Internally, SharedDB employs two different storage engines: Crescando storage manager and a single-threaded key-value store.

*Crescando* [24] is a scalable distributed relational table, designed to provide predictable performance for unknown or dynamic workloads. It is push-based and it batches queries and results, so it perfectly fits the SharedDB framework. Crescando tables do not use indices on relations: instead, full-table scans are performed for each batch, using the *Clock Scan* collaborative-scan algorithm, and indexing query predicates to speed up the tuples filtering process via query-data joins.

The key-value storage engine is designed specifically for SharedDB, in order to support transactional workloads with low latency results. It is less predictable than Crescando. It consists of a dynamic array of tuples and a set of B+ Tree indices. It is a single-threaded operator that performs operations in batches.

Both storage engines provide Snapshot Isolation (SI) consistency. Further work has been done to equip SharedDB with transaction support at the storage layer, but the description of its transaction manager is out of the
2.4. SharedDB and Crescando

scope of this document. Finally, a new parallel MVCC key-value store has been recently implemented for FBX, which exploits lock-free data structures (BW-trees) to minimise the costs of concurrency; it provides a much more scalable solution for transactional workloads and deprecates the previous naïve key-value store.
Chapter 3

System Distribution Design

This chapter describes the design of the distribution of FBX. First, we present some background on how the query plan is constructed and propagated in the non-distributed system, in Section 3.1; the principles of our design are stated and the different design options explored in 3.2. The following Section (3.3) is a discussion of several aspects of the implemented design, some of its challenges and non-trivial optimisations. The enhancements to the design in a ‘multiple query engine’ scenario are depicted in 3.4 and – eventually – some current and future directions of the design of FBX are briefly illustrated in 3.5.

3.1 SharedDB Query Plan

One of the main design decisions of SharedDB is to employ a global query plan, which is shared among all the queries being executed at the same time. This feature enables other peculiar traits of SharedDB, such as batching of queries and result data, shared execution and an always-on core-affine operator model, which would be much harder – or even impossible – to implement in a query-at-a-time system without a shared query plan.

In the current implementation, queries are represented as an operator tree, in which each node is an operator and the links point downwards to its producers (operators that generate the results that it needs to process) and upwards to its consumers (the operators interested in the results it produces). The root of a query, on the server-side, is always a NetServerQuery: NetServers are the entry-points (front-end operators) of SharedDB and are responsible for creating query batches, materializing the results and communicating with the clients; a single query plan can have multiple front-ends for different queries or query classes or employ a single entry-point. The leaves of the tree are storage engine queries, i.e. queries (selections, projections, index look-ups) directly performed in the storage engines (Crescando and
KV store). Any other data processing operator queries (join, group-by, sort, match, limit, etc.) are inner nodes in the tree. This completes the query ADT representation that is seen by the server operators. When an operator processes a query (in one of its batch cycles), it reads its content and sends the subqueries to the correct producers (child operators); then it waits for their results to be pushed up into its result queue and starts processing them. For a more detailed explanation of the operator model and the details of every single operator, see [31].

The global plan is currently shared with the clients: the client has complete knowledge of the list of operators available in the server; in order to send a query, the client uses the operator and workload knowledge to generate the query tree (in a workload specific code module that we call workload driver, which is shared between clients and server); next, it serialises the query and sends it to the server, which conversely deserialises the tree, performs some minimal manipulation of it and executes it. This solution is suboptimal: it causes big efforts when a new workload needs to be ported to SharedDB and brings too much knowledge of the system into the clients. As mentioned in 3.5, the implementation in FBX of a SQL-compliant query compiler and a richer database catalogue on the server-side will eventually hide the server internals and provide the system with complete physical and logical data independence. As far as FBX distribution is concerned, we designed it to hide the decoupling and distribution from the clients, which are unaware of whether the server is completely deployed on a single server or instead is spread across several machines.

### 3.2 Principles and Design Space

The distribution design is based on a set of requirements and abstract principles, some of which are borrowed from SharedDB:

- **query batching**: queries move from one operator to its producers in batches, in order to enable work sharing; queries should be batched also over the network, to achieve higher throughput (thanks to fewer small messages) and simplify the task of reconstructing the batches on the storage side;

- **result batching**: results are also moved and processed in batches; the distribution submodule should also communicate in units of batches, for the same reasons as above;

- **push-based execution**: operators communicate with both their consumers and producers via queues; in the same way, no operators should pull results from the network (by polling), but instead results should be placed in the correct operator queue when they are received;
3.2. Principles and Design Space

- **access & location transparency**: the decoupling of query engines from storage engine should be invisible to the clients, who should create queries as if the system were all on a single machine;

- **replication & partitioning transparency**: similarly, any replication and partitioning of tables across machines should be invisible to the clients, that must reason in terms of a single storage interface;

- **scalability of query engines**: a single storage engine should be able to support 10s to 100s of query engines, potentially running different workloads on the same data concurrently;

- **scalability of storage engine**: a single storage engine should be able to store 100s of tables.

While replication and partitioning are not in the scope of this thesis (a few words about them are spent in 3.5), all the other requirements have to be dealt with in here.

The scalability of the storage engine can be achieved by fragmentation and partitioning; only fragmentation (placement of different tables on different machines) is taken into account in the design below. The scalability in terms of number of query engines depends on how well data is moved between storage engine and query engines:

- **complete result batching** comes with a tradeoff in this scenario: sometimes it is convenient to group results per-machine, before sending, sometimes it is better to broadcast all the results to a full subset of query engines; a detailed discussion about it can be found in 3.4;

- **replication, partitioning and fragmentation** can help in splitting the load onto different machines, in case the outgoing bandwidth of some storage machines is saturated.

Several design options have been considered:

*Overridden operators vs. Stub operators*: a possible design would be to modify the existing operators in order to be aware of the distribution and override the enqueuing/dequeuing operations. Despite it being a possible solution, it would require more engineering effort than adding operators whose only task is the reliable transfer of queries and data from one machine to another. *Stub operators included vs. excluded from query plan*: it would be trivial on the server-side to handle stub operators if they were embedded in the query tree; however, this would imply that the client knows the existence of stub operators (less transparency) or that the entry-point operators perform a major rewriting of the query at deserialisation time. Our design hides the distribution from the clients, which build queries as if all operators were on the same machine, and performs some minimal rewriting of the query, supporting even multiple levels of distribution (e.g. a possible distribution of the query engine into multiple machines) with minor changes to the code.
Per-machine stub operators vs. Per-table stub operators: this decision is intermingled with the next one: if the stub operators have threads and work in batches – at least on one side – the batches of all tables would be synchronised by the stub operator, which would degrade the performance in case of very different table sizes; since the stub operators on the storage engine side need to work in batches (to perform message reordering, see 3.3.3), a per-table stub operator is preferable.

Stub operators own threads vs. Stub operators are callbacks: giving stub operators some cores, like any other operator, seems to be a reasonable solution, but it has some scalability drawbacks: if we have a database with a big number of tables, query engines would need to allocate resources for each table individually, which is an important scalability limitation on the query engine side; instead, having the stub operators as callbacks on the query engine side would solve this problem.

Having analysed the main dimensions of the design space and justified our decisions, we are now ready to describe the full design in the following section.

3.3 Stub Operators Detailed Design and Implementation

Stub operators are the modules responsible for the reliable delivery of queries and results across machines, between standard FBX operators (namely, data processing operators on the query engine side, as consumers, and storage operators on the storage engine side, as producers). Their interface is the same as any other FBX operator:

- `enqueueSingleQuery` and `enqueueBulkQueries`, which are used by consumers to enqueue queries to producers; we will refer to this family of methods simply as `enqueueQuery`;
- `consume`, which is used by producers to send results to consumers.

Although they look like standard operators from the outside, their internal behaviour is very peculiar. We will now describe their details, differentiating between the two types of stub operators: the `StubClientOperator` and the `StubServerOperator`.

The `StubClientOperator` is the stub operator instantiated on the query engine side. It is a thread-less operator, whose operations are performed in the context of function calls, accounted to its consumers (in one direction) and to the networking layer (in the opposite direction).

For its consumers, it behaves exactly like a storage engine operator: when a query needs to be sent to a storage engine located on a different machine, the consumer calls `enqueueQuery` into the stub client; the stub client saves
the query information into its internal structures (a map of query IDs to queries), serialises it and gives it to the networking layer to be sent. In order to abide by the principle of query batching, this operation is always done in batches and full batches of queries are serialised and sent together. Results follow the opposite path through the stub client; since the stub clients don’t own any threads, the result processing needs to be done with the computation resources of some other module: the solution chosen here is to have the networking layer execute a callback whenever a result for a stub client is received. The callback needs to:

- reorder messages that could have arrived in the wrong order from the network;
- restore pointers internal to the result (which would now be meaningless, because the result has a different position in memory with respect to where it was stored on the remote machine);
- update internal batch information: at the beginning of a batch (signalled by the stub server with a special message), the stub client learns which are the consumers for that batch; when it receives results, it sends them to the consumers of the batch they belong to and it removes queries from its internal map, when they are completed.

The StubServerOperator is the stub operator instantiated on the storage engine side. It has its own thread(s) and – differently from the stub client – its queue management follows the same cyclic workflow of the other operators. When some queries are received from the network, the networking layer runs a callback that deserialises them and puts the queries in the operator’s queue. The stub server works with queries in batches: sends the queries to the correct producer, signals the start of the batch to the stub clients and waits for the results. Results are retrieved from the operator result queue (in case there is no resource sharing between operators), each result batch is given a unique sequence number, used for reordering, pointers in the results are manipulated – to make the restore procedure on the client side possible – and finally results are given to the networking layer to be sent remotely.

The query and data flow is summarised graphically in Figure 3.1; beware of the over-simplification of the result path on the stub client side: results are only sent straight to consumers if they come in the right order; otherwise they remain in the stub client until their predecessors in the sequence show up.

The following subsections give further details about some important aspects that have been only briefly mentioned here and deserve more explanations.
3. System Distribution Design

Figure 3.1: Design of FBX distribution: stub operators. The query route is drawn in purple, while the data route is orange.

3.3.1 Networking Layer API

The networking layer – *NetworkManager*, from now on – is the software module responsible for the actual transfer of queries and results from stub clients to stub servers and vice versa. While each table has its own stub operator, the network manager is only one per machine. Its implementation details will be covered in depth in Chapter 4; here we focus on its interface and its interaction with stub operators.

On the sending side, two main functionalities are exposed:

- *buffer acquisition*, through which stub operators can obtain buffers to write their serialised messages into;
- *buffer transmission*, to actually send a message: such a call is non-blocking and the strategy for the asynchronous cleanup of the buffer can be specified as a parameter in the call.

The buffer management is performed by the network manager on the sending side to enable some potential optimisations discussed in 4.3; however, the sender is not obliged to stick to this mechanism and can manage its own buffers, without having to interact with the network manager for buffer
acquisition or buffer cleanup (we never have such a situation in our code, though).

On the receiving side, the network manager exposes the following services:

- *buffer reception*, which is performed internally by the network manager; stub operators register a `handleRecv` callback at start-up: once a message is received, such a callback is run by the networking layer, in the spirit of *active messages*;

- *buffer release*, to give the message buffer back to the network manager, once it is not used anymore by operators.

The network manager owns some threads; it uses them to perform the transfer, check for received messages and run the callbacks. Such a design is more scalable than having each stub operator poll the network for messages to process.

### 3.3.2 Result Structure: Translation and Compaction

Results are generated by storage engine operators, that copy records from their internal datastore into a new result buffer, before sending them to their consumers; for each tuple, they also generate some metadata and the query ID bitset (used to identify the queries interested in the specific tuple).

![Figure 3.2: Structure of a result batch.](image)

The structure of a standard result is represented in Figure 3.2. It comprises two parts:

- the *tuple section*, which contains the fixed-sized metadata; it contains the pointers to the record, to the query ID bitset and possibly other query-type-specific data;

- the *data section*, in which the records and the query ID bitsets are interleaved; the size of the bitsets is batch-dependent, while the size of the
record is schema-dependent.

Firstly, since the tuples contain pointers (not offsets), they need to be translated on the receiver side when sent to a remote machine, in order to match the new position in memory of the buffer. This is solved by translating the pointers into offsets in the stub server, then back into pointers in the stub client. There are possible optimisations to this technique (e.g. only translate on the client side, by sending the memory position of the buffer on the source machine as additional information in the message); Another (cleaner) solution would be to have only offsets everywhere in the code, but this would require a significant code refactoring.

Secondly, the two sections of the result batches are usually not contiguous in memory; even if they were allocated as a single buffer, there would still be a gap between tuples and data, in case not all the tuples slots were used (i.e. the batch is not full). In the general case, the stub server needs to compact the two sections into a single buffer without gaps, to save some network bandwidth.

In Subsection 3.3.4, we will see that both the cost of translation and of compaction can be avoided – in the common case – by specializing the result structure.

3.3.3 Message Reordering

Although the underlying networking protocol (RDMA RC) ensures in-order delivery, such a property is lost in higher layers (MPI and multi-threaded network manager), so we need to restore it at the stub-operator level. Ordering of messages is important for a number of reasons: storage engines can be asked to provide a sorted selection of records; even when no ORDER BY clause is specified at the storage layer, ordering is needed for ensuring that the end-of-result tuple is processed after all the results for that batch.

Message reordering is implemented as a joint effort between stub server and client and makes use of sequence numbers. In particular, the stub server tags each message with two numbers: batch number and sequence number. The batch number is a unique (not necessarily sequential) ID for a batch. When a new batch starts, the stub server sends a START_BATCH message to the stub client, specifying the queries that belong to that batch; thanks to this message, the stub client can identify which are the consumers of that batch. The sequence number is a sequential unique ID of a message in a batch. The stub server ensures that the messages are tagged in order; the stub client works as a state machine, processing messages only once all the messages with a lower sequence number have been processed (storing them for later consumption, otherwise). The last message of a batch is tagged by the stub server as END_BATCH message, to trigger the batch cleanup on the stub client side.
3.3. Stub Operators Detailed Design and Implementation

While message tagging at the stub server is trivial in a single-threaded implementation, it is slightly more involved in a multi-threaded scenario; we briefly discuss about that in 3.3.4.

3.3.4 Optimisations

In order to keep the thesis at a reasonable size, we won’t discuss all the implementation details of the stub operators; we will focus instead on a couple of optimisations that play an important role in the overall design of the system: tuple-free results and parallel stub servers.

The first optimisation concerns the way in which stub servers send results. If we send all results as described in 3.3.2, we pay a very high price in stub operators:

- iterate over all tuples for pointer-offset translation (both in the stub server and the stub client);
- memcpy of all result batches for compaction (in the stub server).

This price mostly affects results of type SELECT, which are usually the most voluminous.

A couple of observations lead us to a solution that avoids this overhead:

- the metadata in the tuple section only contains the pointers to query bitsets and records for select results;
- the bitsets and records are generated sequentially by the storage engines.

This means that the pointers to bitsets and records can be easily inferred as a linear displacement from the beginning of the data section as:

\[
\begin{align*}
RecordPtr_i &= RecordPtr_0 + i \cdot (bytes_{QueryBitset} + bytes_{Record}) \\
QueryBitsetPtr_i &= QueryBitsetPtr_0 + i \cdot (bytes_{QueryBitset} + bytes_{Record})
\end{align*}
\]

and the tuple section doesn’t need to be sent over the network at all. Such an optimisation is critically important for the Crescando storage engine (which produces most select results), where – depending on the schema and the workload – up to 50% of the bandwidth can be saved.

In the second optimisation, we remove a potential bottleneck in the stub server. In all the previous discussion, the stub server has been assumed as a single-threaded operator; while such an implementation is simpler, it has performance problems, in particular if all tuples need to be scanned in the stub server for translation.

The solution, already adopted in many other operators, is to make it parallel: a ParallelOperator is a set of worker threads that perform a job and are coordinated by a master thread. The operator has a single result queue and
query queue: getting new queries and starting new parallel jobs is a responsibility of the master thread; once a job is started, worker threads can get result batches concurrently from the result queue, process them and send them to the consumers in a parallel fashion. This enhancement gives us also more resource allocation freedom: parallel operators can be individual entities with their own resources, but can also be grouped together in OperatorGroups. In the latter case, all the group shares the same pool of computing resources; moreover, the result queue between two operators belonging to the same operator group is removed and substituted by a cheaper function call.

The activity that is performed by the workers is exactly the same as described in the previous sections; the only noteworthy change is that message tagging needs to be synchronised among workers: this is achieved through locking, so that getting a result batch from the queue (or up-calling the stub server, in the case of operator groups) is performed atomically with getting and increasing the sequence number.

3.4 Multiple Query Engine Design

The design described above can be adapted in a trivial but non-optimal way to a multi-query-engine scenario. The following technical aspects need to be addressed:

- the storage engine needs to decide which query engines should receive each result batch;
- the storage engine must be able to uniquely identify queries that are sent to it by several query engines.

The following subsections deal with each issue individually.

3.4.1 Result Multicasting and Filtering

The first problem can be initially solved in a simple way: each batch is fully received by all machines that have queries involved in that batch. This is in the spirit of what SharedDB is doing for the non-distributed case at the operator level: all operators that have a query in a batch receive all results for that batch and filter them on their side. Although this can work locally, it becomes a major scalability bottleneck when distributing the design.

A solution to this problem has been designed and partially implemented; it involves a hybrid mechanism with both multicasting and filtering in the stub server. The two considered techniques are:

- multicasting is the best solution when many query engines are interested in the full set of tuples in a batch: in such a case filtering results

\footnote{This is the technique currently implemented in the stable version of FBX.}
3.4. Multiple Query Engine Design

for each query engine would require the generation of many different replicas of the same results (unnecessary memcpy’s). Workloads that perform at least one full scan per batch (e.g. analytical workloads like Star Schema Benchmark and TPC-H, see 5) would benefit from such a mechanism; furthermore, some bandwidth can be saved on the storage side, by exploiting some hardware multicasting support or some clever broadcasting algorithms; details about the (partial) implementation of multicasting are discussed in 4.5;

- filtering can be instead employed when many query engines access a table for highly selective queries: in that case, it is better to create different buffers for different machines and send the selection to each machine individually.

Given the usage patterns of different types of storage engine and the current work in developing a federated storage in which Crescando mostly receives low-selective scan queries and KV stores receive point queries and updates, in a first implementation Crescando would perform multicasting, while KV Store results would be filtered per-machine. This solution is not optimal: with the presence of an optimiser and of selectivity statistics, individual engines could be able to distinguish when to perform multicasting and when filtering, on a per-batch basis.

3.4.2 Query ID Space Management

Query IDs are used by the operators to identify queries themselves and as tags of intermediate results to determine which queries can be interested in specific tuples (SharedDB’s data-query model): the query bitset of a tuple contains the information of all the queries interested in the tuple. Since bitsets associate each bit to a query in incremental order, it is in the interest of the system to keep the query ID space as compact as possible (to have small bitsets).

While the query ID management is trivial in the case of a non-distributed system or a single-query-engine scenario, where IDs are acquired in the front-end operators and the ID management is centralised and synchronised among NetServers, the problem is more complicated in the case of multiple concurrent query engines. Many design options have been considered:

- centralised query ID manager: a single module has the right to associate IDs to queries; in this case each query engine would have to pay the cost of calling (remotely) the ID manager for each query; on the other hand, the bitsets would be optimally compact;

---

2Multicasting has been implemented and partially integrated in the full system; filtering hasn’t been implemented: it requires the KV store to allocate multiple buffers internally, store results in each of them depending on the source of different queries and tag the batches with the ID of the machine to specify to the stub server which engine should receive the message.
3. System Distribution Design

- **Static** query ID partitioning: at start-up time each query engine is given a portion of the query ID space and it will manage it locally; it is the most scalable solution, but it can produce very sparse bitsets, depending on the implementation;

- Query ID *translation* in the stub server: query engines associate IDs that are only locally unique, but can clash with other remote query IDs; the stub server would reconcile the IDs by translating the IDs of the incoming queries to remove collisions, maintain such a per-batch mapping and translate the bitsets of the outgoing results back to query-engine-local IDs; this solution increases the complexity and the cost of the stub server and would jeopardise other design decisions, such as multicasting.

The adopted solution is a modular static partitioning of the query ID space: each machine is given a *QueryEngineID* at start-up and its query ID space is constituted by all IDs of the form:

\[ QID \equiv \text{MachineID} \pmod{\text{MaxMachineID}}. \]

This solution has proved good enough for our requirements so far, but not many tests have been performed about the system scalability. It is possible that – in case of high inter-workload variety – the used query IDs are very sparse (because the number of queries coming from different query engines is very diverse): potentially, a more dynamic solution could be needed in the future, e.g. with a centralized manager that gives batches of IDs to query engines, that would adapt their ID space depending on the current load.

3.5 Current and Future Work

Some work that has been done simultaneously by other team members (Claude Barthels, Jana Giceva, Darko Makreshanski) includes:

- Development of a new interface to the storage engine, that unifies the API for different types of backends: it allows easier replacement, partitioning, movement, replication by making any call to the storage engine transparent on the query engine side;

- Implementation of a parallel multi-version key-value store, that replaces the single-threaded KV store;

- Replication of tables: it is now possible to have a single table in several (homogeneous or heterogeneous) backends; a single replica is designated as master and is the only one that receives updates; updates are propagated to the other replicas in the background;
3.5. Current and Future Work

- persistence: logging to secondary memory has been implemented; it is also possible to snapshot the whole database, in order to stop it and then resume it from its previous state.

There are various improvements and enhancements to the system that are planned for the future:

- table partitioning: it will be possible to specify a partitioning strategy of a single table on multiple machines;

- result projection: the storage engine should transfer remotely only the fields that are required by the parent queries; this is particularly useful for analytical workloads, which are very bandwidth-sensitive;

- query compiler: now all queries of different workloads are manually matched to the global query plan, which is hard-coded in the workload driver; a SQL-compliant query compiler would make the system less vertically integrated, more maintainable and easier to extend with more workloads.
Chapter 4

Networking Layer Implementation

In Chapter 3, we have described the general design and implementation of FBX’s decoupling and distribution, without entering the details of how the message transfer is performed and treating the networking layer as a black-box providing a message-passing semantics with receiver callbacks (active messages). In this chapter, we will dive into the networking layer, providing information about design and implementation decisions, backed up by micro-benchmarks, whenever possible. The impact of the mechanisms discussed in 4.5 are analysed in Chapter 5, in action on real workloads.

4.1 Choice of MVAPICH MPI

We have already described in Chapter 2 what primitives (verbs) are exposed by RDMA and how those are used internally by MPI. When we had to decide whether to implement our own library on top of pure RDMA or to use a third-party library, we chose the second option for several reasons:

- common MPI implementations contain most of the mechanisms needed to implement a fast networking layer on top of RDMA, without ‘reinventing the wheel’, in terms of eager and rendez-vous protocols and API semantics (channel-based and RMA);
- they are constantly maintained, tested and updated by a large community.

Since we wanted an open-source MPI version that supported InfiniBand, our decision was narrowed down to either OpenMPI or MVAPICH. OpenMPI doesn’t support multiple threads calling into the library at the same time: we considered this limitation quite strong for the design of the networking layer, so we picked MVAPICH.

Using MPI comes with some cons, that we can describe a-posteriori and would make us retreat – at least partially – from our early choice:
4. Networking Layer Implementation

- MPI abstracts from the physical channel almost completely and doesn’t give the programmer enough freedom to exploit some key underlying protocol features; for instance, RDMA Service Levels are hidden, making message prioritisation much harder to achieve, and transport mode (UD, RC) cannot be chosen on a per-flow basis;

- some mechanisms are very advanced (e.g. software broadcast algorithms) but they come with the cost of a rigid API, hard to use in a non-trivial setup; for example, broadcasting is impractical if the message size is unknown on the receiver size and broadcast groups are not dynamic (cannot be changed from message to message);

- the library has internal book-keeping, message-tag matching and other hidden mechanisms that make it very generic, but expensive when the design of the user system doesn’t utilise some of these features or could override them in a simplified and more efficient way;

- given that the MPI standards specify the interfaces, but not the implementation techniques, any details are implementation-dependent, mostly undocumented and buried in the code; this transforms the analysis of the performance of the system into a non-trivial process, where MPI is a black-box, quite hard to model, measure and customise.

In a way, this discussion comes back to the everlasting tradeoff between abstraction and performance: abstraction is a powerful principle to make a system easy to use and understand, but comes with a big cost for applications that need more low-level power.

4.1.1 Basic Benchmarks

To measure MVAPICH performance, we have used the standard OSU MicroBenchmarks [3]. Throughout this chapter, we will focus on throughput, since our system is designed around batching and computation sharing with a high number of clients and a ping-pong latency benchmark in an underloaded environment would provide very little information; we will show latencies when discussing the performance of full-system benchmarks, in Chapter 5.

Figure 4.1 presents the performance of MVAPICH send/recv with respect to raw RDMA InfiniBand Write, on a 6 GB/s connection. The machines used in this experiment (and in all the following ones) have an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, 512GB DRAM, a Mellanox ConnectX-3 Single Port [with VPI] MT27500 Family HCA and are inter-connected by a Mellanox SX6018 FDR switch.

The plot shows the performance of the multiple transport mechanisms that can be used for rendezvous protocol: an RDMA-Read-based solution, one
4.1. Choice of MVAPICH MPI

![Figure 4.1: Bandwidth comparison between MVAPICH implementation and raw InfiniBand.](image)

that exploits RDMA Write and one that internally uses channel semantics. We can see that Read and Write perform almost equally well, reaching the full bandwidth for big messages (~100KB), while the send/recv implementation performs poorly. All the MPI schemes are losing a non-negligible amount of bandwidth for small messages: this is due to the internal mechanisms of MPI we were mentioning before (message matching, book-keeping and data structures update, etc.).

The OSU Benchmark measures the bandwidth for each size of messages independently in the following way: a sender node sends a window \( N \) messages of the same size to a receiver node and then waits for an acknowledgement (single fixed-size message); this is repeated for \( K \) steps: after \( S \) steps are skipped, the time duration of the remaining \( L \) steps is measured (where \( K = S + L \)). The bandwidth is then calculated as ratio between the amount of data moved in \( L \) steps and its duration.

We have implemented a similar micro-benchmark to measure the performance of our layer on top of MPI. The MPI_Send and MPI_Recv calls have been substituted by calls to our library functions; moreover, our benchmark supports multiple sending and receiving threads (synchronised by barriers), mimicking more closely the behaviour of a real application. In all following sections we will make use this micro-benchmark, unless stated otherwise.
4.2 Big Messages Management

Big messages are transferred in our system using the point-to-point send/recv semantics of MPI. An application thread, with a message ready to be sent, calls the send method of our library, that inserts the message into a queue and returns immediately to the caller. The buffer is then taken by a network layer thread, which sends it over the network: initially the implementation was using a blocking MPI_Send, then substituted by a non-blocking MPI_Isend. In the case of non-blocking operations, the buffer transmission is completed only when an MPI_Wait is called on the request handle: in our implementation, we fire an MPI_Waitall on all pending requests after enough requests have been accumulated or a timeout is reached (both batch size and timeout are tunable parameters).

On the receiver side, we cannot simply call an MPI_Recv, because our system has variable-length messages and the sizes are unknown to the receiver. The mechanism is then slightly more involved (and expensive): the networking thread probes the channel for new messages in a loop, using the MPI_Improbe call, which returns the size of the message being received and a handle to the message; once a receiving buffer is allocated, now that the size is known, the message can be received by MPI_Mrecv, passing the handle, that is used to match the new (recv) request to the previous (probe) request.

Instead of allocating the buffers for every received message, the networking layer employs a buffer pool. Its usage has several benefits:

- it avoids the cost of dynamic memory allocation on the data path;
- it allows the exploitation of the registration cache offered by MVAPICH, a mechanism in which the memory regions registered for RDMA transfers are kept in a cache and lazily deregistered, letting some buffers be reused for new messages, removing most registration cost from the data path.

Our buffer pool is organised in buffer classes with exponentially increasing sizes; each class is a collection of buffers implemented as stacks; buffers are served with LRU policy, to exploit the time locality of the registration).

In a similar way, the application sender acquires a buffer from the networking layer before filling it with a message and sending it, in order to enable reuse of buffers and caching of registrations also on the sender side.

Figure 4.2 shows how the system performs for messages of different sizes. The result is shown for 4 network threads (2 senders and 2 receivers) and 16 application threads (8 senders and 8 receivers); the same results are achieved for different combinations of threads, with negligible differences. While the performance is satisfactory for big messages, it degrades quite heavily for small messages with respect to the pure MPI benchmark. In 4.2.1 we look for
4.2. Big Messages Management

4.2.1 Issues with Point-to-Point Semantics

To find the source of the poor performance for small messages, we have followed a step-by-step process: starting from a pure MPI benchmark (that uses \texttt{MPI\_Send} and \texttt{MPI\_Recv} directly), we have added incrementally more and more features of our networking layer on top. First, the sender mechanisms have been added and we have tried multiple optimisation techniques on the sender side: non-blocking calls, removal of sender queue for small messages, tuning of parameters in the benchmark. After such tuning, the sender system was able to perform as in Figure 4.3.

Although these results seem promising, as soon as the probing mechanism is added again to the receiver, the performance drops back to the previous results of Figure 4.2.

The probing system is the main issue in our design: the MPI internal cost of saving and matching the messages, with the help of data structures that are not primarily optimised for extensive multithreading, makes the overhead of such a call unacceptable in our context. Since we cannot give up on variable-sized messages, we can try to bypass most MPI internal structures by exploiting its RMA API, which is the topic of the next section.
4.3 Small Messages Management

A good design for small messages would avoid as much library overhead as possible to achieve lower latency and higher throughput. For this reason, we looked into MPI one-sided operations, that are supposed to be implemented as a thin layer on top of true RDMA.

Our implementation uses a single ring buffer with explicit head and tail pointers, for the communication between each pair of nodes: to a first approximation, the sender sends each message in the format \((\text{size}, \text{payload})\) and updates the head of the stream; the receiver polls the head for changes, reads up until the head and updates the tail. To avoid control message flooding, we update the pointers lazily: the head is updated only after a batch of messages was sent; in the same way, the tail is updated once everything up to the head snapshot has been read and processed.

Concerning the MPI interface, we are using \(MPI_{\text{Put}}\) to perform the RDMA Write, while \(MPI_{\text{Lock}}/\text{Unlock}/\text{Flush}\) is used as a synchronisation mechanism: they let us perform updates treating the receiver as completely passive, without any need for real synchronisation (these calls are not real exclusive locks, but can be interpreted more correctly as sender time windows: \(\text{lock}\) and \(\text{unlock}\), that we call only at start-up and cleanup of the whole system, are calls to start and end a window; a \(\text{flush}\) call is equivalent to closing a window and opening a new one; once \(\text{flush}\) is called, the sender is sure that all the writes of the previous window have completed).

Different mechanisms can be used on the sender side to exploit buffer and
registration caching, with different tradeoffs:

- the networking layer can memcpy the application buffers into its own big buffer, which allows some level of message coalescing (i.e. send one single message instead of many smaller ones), at the cost of more memcpy’s;

- the networking layer gives a pre-registered pooled buffer to the application, that fills it with the message payload and then gives it back to the networking layer, which can send it as it is, avoiding any memcpy.

The performance of each of the systems is compared in Figure 4.4.

![Figure 4.4: Bandwidth of small messages solution (RMA with and without memcpy & coalescing).](image)

We see that memcpy is very effective for very small messages, where the power of coalescing is exploited; for medium-size messages, the solution that uses preregistered buffers and skips memcpy’s beats the previous one. Since we are mostly interested in big and medium messages (we provide batching at the application layer for both queries and data), the second design is more appealing. If we realise that some workloads (extreme latency-sensitive transactional) or some parts of the system (e.g. updates propagation between storage backends, transaction management) need very small messages, we could dig up the coalescing mechanism in the future.
4.4 Hybrid Solution

In the latest version of the networking layer, we fuse the mechanisms described in 4.2 and 4.3, switching from one mechanism to the other at the natural point at which one technique overtakes the other one. The resulting throughput is illustrated in Figure 4.5.

![Figure 4.5: Bandwidth of hybrid solution.](image)

It can be considered a good result for our purposes, since for messages bigger than 10KB we are able to get almost the full bandwidth. For small messages we cannot beat the pure MPI benchmark, because of our added overheads (buffer pool, sender request queue, receiver callback) and locality of buffers (the standard benchmark sends always the same buffer, while our micro-benchmark performs the allocation of buffers with buffer pools, as in the real system).

We also wanted to make sure that messages of different sizes were not fighting for the bandwidth in a destructive way, so we ran a benchmark with two different sizes of messages being sent at the same time and measured the message ratio and the total bandwidth of each type. The results are plotted in Figure 4.6. The bandwidth of big messages is only slightly affected by the presence of small messages; when very different sizes are involved, the bandwidth of the small messages is affected, because the overall message rate is reduced, due to higher per-message transmission times; nevertheless, small messages have a higher relative rate than big ones; all this bestows an acceptable level of fairness on the system. When different prioritisation classes are required, more involved mechanisms need to be implemented on
4.5 Multicast Experiments

In 3.4.1, we discussed how multicast would be used in our design; here we present some preliminary experiments that show the performance of MPI broadcasting. We use a micro-benchmark similar to the one discussed previously in 4.1.1, with the difference that instead of having one sender and one receiver node, we have now a single sender and multiple receivers. The numbers are shown in Figure 4.7, that compares MPI broadcasting (software
4. Networking Layer Implementation

and hardware) to a send/receive solution, that simulates broadcasting by multiple sends.

We see that the software-based broadcasting is superior to the ‘multiple sends’ simulation, for messages of medium and large size, already for 3 receivers. On the other hand, we confirm that hardware-based broadcast exhibits poor performance for a small number of clients. We couldn’t test the scalability with more machines, due to technical problems in the infrastructure (issues in installing MPI correctly on a new cluster of 12 machines).

4.6 Networking Layer Resource Management and Message Prioritisation

This section concerns two topics that are less about the design of the network protocols themselves and more about the creation of mechanisms that can be tuned for a better overall performance of the system:

- resource allocation for the networking library;
- message prioritisation.

In the context of this section, by resource allocation we mean the assignment of networking tasks to different threads. We have implemented 2 possible schemes:

- a specialised threads assignment, in which we have threads allocated specifically for each task (send of small messages, send of big messages, receive of small messages, receive of big messages, execution of callbacks);

- a homogeneous threads assignment, in which all threads perform all tasks in a non-blocking loop fashion (i.e. “if there is something to do for a task, do it, else switch to the next task”).

1 Due to the implementation details of the networking layer, small message handling is accessed exclusively in a locked critical section.
Message prioritisation is important in the case of mixed OLTP/OLAP workloads: if we have both big analytical queries that move all the tables over the network for each batch (throughput sensitive) and small transactional point queries that fetch a minimal portion of a table (latency sensitive), we don’t want the transactional data to be enqueued together with the analytical data, because this would increase unbearably the latency of the transactional stream. To avoid the degradation of transactional workloads due to analytical workloads, we need to implement a prioritisation in the way messages are enqueued.

We have implemented a system that categorises the messages in two classes: latency sensitive and best effort. The two classes use two separate queues (for both small and big messages) and the sender implements the prioritisation policy. At the moment, the policy works as follows: messages are picked from the latency-sensitive queue until it is empty, then the best-effort queue is probed.

This simple implementation has proved to be effective in our benchmarks; if needed, it could be enhanced by balancing the prioritisation (that now is completely biased towards latency-sensitive), parametrising the polling rate from the queues and adding more than 2 priority classes. We postpone the evaluation of this mechanism to Chapter 5, where it is seen at work in the context of a real mixed workload.
4. Networking Layer Implementation

Figure 4.7: Bandwidth for broadcast for 1, 2 and 3 receivers.
Chapter 5

Evaluation

In this chapter we evaluate our design on standard benchmarks. The analysis comprises OLTP, OLAP and OLTP+OLAP mixed workloads, in order to prove that the system is able to handle a variety of different load types, with different network requirements.

After a short description of the benchmarks and of the experimental setup (5.1), a first general overview of the results for each workload is provided in 5.2; next, some more details about the networking overhead are presented in 5.3; the chapter is concluded by a few more benchmarks of some specific aspects of the system.

5.1 Description of Benchmarks and Experimental Setup

The following benchmarks have been considered and results for them will be presented in the next sections:

- **TPC-H** [5] is a benchmark that simulates a typical decision support workload, with heavy business-oriented analytical queries on a realistic wholesale supplier database. The database consists of 8 tables, that are all stored in Crescando in our implementation. The cardinality of the tables is defined by the Scaling Factor (SF), which roughly represents the number of GB the raw data occupies in main memory: each table has a base cardinality, which is multiplied by the SF (except for 2 minor tables). The original workload comprises 22 read-only queries, containing expensive scans, joins and aggregate operations. We have simplified some complex aggregations (since not all SQL is supported in the current implementation of the query engines), by substituting them with a simple SUM; moreover we have removed some queries with complex predicate filters (e.g. predicate disjunction and multi-attribute filters, which are not supported by Crescando) and fully re-
moved some queries that did not make sense anymore after these simplifications: this results in a workload composed by 15 queries.

- **Star Schema Benchmark (SSB)** [23] is another analytical workload, derived from TPC-H. It changes the Normalized TPC-H into a star schema data-mart form, which more resembles the standard denormalised representation in data warehouses. The database consists of 5 tables (a big fact table and much smaller dimension tables). It also defines the concept of Scaling Factor, as in TPC-H. The benchmark comprises 4 query flights, each with 3-4 selectivity flavours. They perform extensive scans, joins and sum aggregations.

- **TPC-C** [4] is a transactional benchmark that simulates an OLTP activity (management, sale, distribution) on a wholesale supplier database. All read-only and update operations are performed as part of database transactions. The database consists of 9 relations, that we locate in our parallel MVKV store. The size of the database is defined by the number of warehouses, which scales linearly with the number of clients/terminals (the number of clients per warehouse is usually a fixed parameter). The workloads comprises 5 transactions, that are quite write-intensive. We run our clients without think time.

- **TPC-CH** [14] is a recent benchmark (not yet fully standardised) created by TU Munich, that tries to merge TPC-H and TPC-C. The motivation for it is simple: since many workloads today are a mixture of OLTP and OLAP that act on the same data, many research and commercial solutions are available that provide support for both analytical and transactional loads; the analysis of their performance with TPC-C and TPC-H alone does not describe well the behaviour of such systems, because such a setup would completely lack a characterisation of the system when both OLTP and OLAP queries are being run on the same data at the same time. TPC-CH takes the TPC-C workload as it is, adds some relations from TPC-H and some of its analytical queries, modified to work on the new schema. The size of the database is again a function of the number of warehouses; the number of analytical queries can be varied independently (creating mixes with different OLTP/OLAP ratios). We run the benchmark with tables replicated in both Crescando and MVKV store, with updates and transactional queries performed on the KV store and analytical queries as Crescando scans.

All experiments will present two metrics:

- the throughput of the system (in queries per hour or transactions per minute);
- the response time of the system (in seconds).
In all graphs, the number of clients (or of warehouses) is the varying factor; more experiments are planned for the future, varying other scalability dimensions (data size, number of machines).

The environment where the experiments have been run is:

- 4 machines with 40 Intel(R) Xeon(R) CPUs [E5-2630 v3 @ 2.40GHz] (although this results in 80 cores per machine, with hyperthreading enabled, we always use only 40 cores, to avoid any degradation due to hardware sharing between hyperthreads), 512GB DRAM over 4 numa regions, a Mellanox ConnectX-3 Single Port [with VPI] HCA;

- 1 Mellanox SX6018 switch.

When running the system in non-decoupled mode, a single machine was allocated to the server; in the decoupled setup, two machines were given to the server, one for the storage engines and the other for the query engine. Depending on the specific benchmark, clients could be integrated in the query engine (e.g. TPC-C) or placed on a separate machine (e.g. TPC-H). Whenever nothing more is specified about decoupled mode, we imply that we are using our standard configuration, in which storage engine tables are in the same operator group as their stub server.

5.2 General Evaluation

5.2.1 Analytical Workloads

Both SSB and TPC-H have been run varying the number of clients up to 2048, with a scaling factor (SF) equal to 100. Figure 5.1 and 5.2 report the results in terms of overall throughput of the system and latency (50th, 90th and 99th percentiles).

The behaviour of the two workloads is similar: for a small number of clients, the benchmarks are mostly network-bound in the decoupled case, which exhibits worse performance than the non-decoupled counterpart (both in throughput and latency); for high number of clients, the benchmark becomes compute-bound in the storage engine, showing now the benefits of decoupling (and of the increased amount of resources allocated for both Crescando and parallel join operators).

The compute-boundness rises from the implementation of indexes in Crescando: even though Crescando is designed to be almost unaffected by the variation of number of clients, its query-index implementation is not scalable enough; in particular, if a query has multiple predicates, only one predicate is indexed, while all the others need to be computed for each tuple that matches the indexed predicate. By increasing the number of clients, also
5. Evaluation

Figure 5.1: Throughput and Response Time for Star Schema Benchmark, SF 100.

the number of un-indexed predicates to be computed increases, causing an overall slow-down of the storage engine.

The compute-boundness argument explains the difference in peak throughput performance between TPC-H and SSB: although TPC-H has a richer range of queries, SSB has a specific query (Query1) that generates heavy processing in Crescando (due to its 3 inequality predicates), while TPC-H has no such heavy queries predicate-wise (the number of predicates in TPC-H is always at most 2, most of the times on the same field, which lets Crescando use a range index, without leaving any predicate un-indexed). For this reason, SSB has lower performance than TPC-H; we have measured that substituting Query1 with some lighter queries, Star Schema Benchmark is
Another interesting observation is that the number of clients at which the decoupled mode becomes better than the non-decoupled mode is different in the two benchmarks. This can be explained again by the different network- and compute-bound nature of the workloads: TPC-H moves considerably more data per-batch than SSB; both the Crescando cycle and the amount of data moved increase with the number of clients, with Crescando having a faster increase, in particular for SSB. Due to the previous effects, SSB becomes compute-bound faster than TPC-H, which explains why the decoupled mode overtakes the non-decoupled with fewer clients in SSB than in TPC-H.
5. Evaluation

5.2.2 Transactional Workloads

TPC-C has been run with a variable number of warehouses, from 1 to 500. The plots in Figure 5.3 show the results for a number of clients that is 5 per warehouse; in other words, the number of clients varies from 5 to 2500. Also in this section, we plot the end-to-end throughput and the latencies in 50th, 90th and 99th percentile.

![TPC-C Throughput and Latency Graphs](image)

**Figure 5.3:** Throughput and Response Time for TPC-C Benchmark, 5 clients per warehouse.

Differently from the analytical workloads, which are mostly bandwidth sensitive, TPC-C is very latency sensitive. This is a potential challenge for our system, which is based on extensive batching and resource sharing between queries, i.e. sacrifice some latency to the benefit of throughput. A compar-
ison to the results of other systems is out of scope here, while we focus on
the internal comparison between the decoupled and non-decoupled mode.

We see from the plots that the non-decoupled mode performs always better
than the decoupled. In this experiment setup, no resources are gained by
the storage engine once decoupled, because the amount of cores allocated
to the query engine is negligible (most transactions are constituted by point
queries and updates, with rare 2-table joins). This implies that the most
visible effect of decoupling is the latency added by the networking layer.

The performance of our system is very affected by the increased number
of warehouses, particularly at its 90th and 99th percentile: most of this
degradation is independent from the network, since it is present also in
non-decoupled mode; nevertheless, the overhead due to networking is not
negligible. We will analyse this effect more quantitatively in 5.3.2.

5.2.3 Mixed Workloads

TPC-CH has been run with a fixed number of warehouses equal to 100. The
number of transactional and analytical clients has been independently var-
ied and the results – in terms of analytical throughput (queries per hour)
and transactional throughput (transactions per minute) – are plotted in Fig-
ure 5.4. In the experiment presented in here, a single storage machine was
used for both KV store and Crescando scans, with replication, propagation
of updates and prioritisation (see 4.6) enabled.

The results cannot be directly compared to the ones in the previous sections
(the analytical part of the workload is quite different from TPC-H; the trans-
actional part is the same as TPC-C, but the setup is different and the current
experiment is run with a fixed number of warehouses). Nonetheless, we
can make some good points about the behaviour of the system when two
different workloads are run together: the transactional throughput is almost
not affected by the number of analytical clients that run concurrently; on
the other hand, the analytical workload is slightly affected by the number of
transactional clients, due to the replication mechanisms.

We omit an analysis of the effect of distribution of the storage engine and of
the replication mechanisms, since they are not a contribution of the thesis;
instead, we will describe the effect of message prioritisation in 5.4.2.

5.3 Network Overhead Analysis

5.3.1 Analytical Workloads

We have designed another experiment to isolate the effect of the network
from other factors that change when we decouple the storage from the query
5. Evaluation

![TPC-CH: Transactional Throughput](chart1)

![TPC-CH: Analytical Throughput](chart2)

**Figure 5.4:** Throughput of transactional (top) and analytical (bottom) workloads in TPC-CH Benchmark, 100 warehouses. Both the number of transactional and analytical clients are varied in both graphs.

The configuration of the new experiment is different in two aspects from the previous experiment of Section 5.2.1:

- operator groups are not exploited in the non-decoupled mode and data processing operators are given a set of cores that is completely separate from the one given to Crescando tables; in the previous experiments, the non-decoupled mode was assigning all tables and join operators to the same group;

- the number of cores given to each storage and data processing operator is the same in non-decoupled and decoupled mode; this is different...
from the previous experiment, in which the double number of cores due to decoupling was exploited by giving more cores to Crescando and parallel join operators. More concretely, out of the 40 cores present on a machine, 20 cores are allocated to the storage engine and 20 to the query engine, in the new experiment.

The result of the new experiment is shown for both SSB and TPC-H in Figure 5.5 and 5.6. Similarly to the previous experiment, we see that the throughput is affected by the network for small clients, in which the overhead due to the network-boundness is more evident; for many clients, the performance of the two modes is almost equal, since in both cases Crescando becomes
the main bottleneck and the network overhead becomes negligible, due to pipelining in processing results.

The increased latency for few clients is mostly due to the network bandwidth bottleneck: we see a mean bandwidth of 3.5-4 GB/s being transferred over the network, with peaks of 5 GB/s, which is the maximum throughput that our library can provide for big messages.

Without the help of additional resources, the decoupled mode is not able to perform better than the non-decoupled mode. Nevertheless, system statistics show that the Crescando cycles are shorter in the decoupled mode, because Crescando tables don’t share any memory bandwidth with query en-

**Figure 5.6:** Throughput and Response Time for TPC-H Benchmark, SF 100, isolating networking overhead.
5.3.2 Transactional Workloads

In the experiment of 5.2.2, a direct comparison of the mean latencies of decoupled and non-decoupled mode shows that 70-95% of the latency of the decoupled mode is due to effects also present in the single-machine system. We are not interested in the examination of KV store implementation-specific issues; in the next paragraphs we focus on the remaining 5-30% of the latency, that is caused by decoupling.

In order to avoid to consider aborted transaction in our analysis, we analyse the latencies for the configuration with a single client per warehouse, where aborts are almost non-existent.

The difference in mean latency that we measure (between non-decoupled and decoupled mode) ranges from 2ms to 15ms. The mean critical path number of queries per transaction is 13.7; this number is the mean of the number of queries for each type of transaction, weighted by the relative presence of each type in the mix; we count the number of queries in the critical path of each transaction because some queries are run simultaneously in the same transaction. Hence, we have a per-query overhead of 150us to 1ms.

Ping-pong latency micro-benchmarks show that InfiniBand by itself is able to achieve latencies in the order of 1-2us (for small messages of a few KB, which is comparable to the size of messages we send in TPC-C); on the other hand, for the same sizes, MPI shows latencies up to 1 order of magnitude higher.

We have just estimated an overhead that is 2 orders of magnitude bigger than raw InfiniBand; many factors influence this performance gap: our system is measured under a real workload, which stresses the RDMA connection differently from a ping-pong micro-benchmark (which waits for a reply before sending another message); our implementation of the networking layer contains a few concurrency-related overheads (e.g. buffer pool, sender queues), which can be further optimised, but will never become costless; MPI adds some overhead also in RMA calls, even if smaller than for point-to-point operations; in the homogeneous-threads implementation of our layer, small-message sender operations are performed in a loop, together with small-message receiver and big message operations: this increases dramatically the worst case latency (a message could need to wait for a full cycle before being sent). The previous points explain also the latency growth when the number of warehouses is increased, due to higher costs of concurrency and greater load on the network.
5. Evaluation

5.4 Further Benchmarks of Specific Features

5.4.1 Resource Management

We have measured the different performance of SSB in the presence of separate threads or homogeneous threads, the two resource management techniques explained in 4.6. The experiment was run with scaling factor 10; Crescendo tables share the resources with their stub servers and are assigned all the cores.

![Throughput and Response Time for Star Schema Benchmark, SF 10, for different resource management techniques.](image)

**Figure 5.7:** Throughput and Response Time for Star Schema Benchmark, SF 10, for different resource management techniques.

The results, depicted in Figure 5.7, show that there is no impact either in
5.4. Further Benchmarks of Specific Features

throughput or in latency, switching from one technique to the other. Moreover, we have seen that our system is able to perform well in TPC-C with a single homogeneous thread. Given the flexibility and simplification that homogeneity provides to our resource management, this solution is preferred, whenever possible. If future benchmarks provide us with the evidence of a bias towards a particular type of message (small or big), the problem can be addressed by restoring the separation of tasks into different threads.

5.4.2 Message Prioritisation

![TPC-CH: Prioritisation effects on Transactional](image)

![TPC-CH: Prioritisation effects on Analytical](image)

**Figure 5.8:** Throughput of transactional (top) and analytical (bottom) workloads in TPC-CH Benchmark, 100 warehouses, comparing the results with and without prioritisation enabled. Analytical results are shown for 100 analytical clients, while transactional results for 500 transactional clients.
5. Evaluation

In Figure 5.8, we can appreciate the effect of prioritisation on the throughput of analytical and transactional workloads. The experiment is run with the same setup as in 5.2.3; when plotting the transactional throughput, the number of transactional clients is fixed to 500; for the analytical throughput, the number of analytical clients is 100.

When no prioritisation is present, the latency sensitive messages belonging to the transactional workload are enqueued together with the analytical ones; this generates a bias against the transactional workload, because the mass of the analytical stream is bigger and saturates the channel easily. Enabling prioritisation, the transactional throughput increases of more than 3x, thanks to its dedicated sender queue; also the analytical workload is affected by this change, in opposite direction, but the degradation is much more restrained (20-25%).

The results are influenced by the lack of power on the underlying RDMA prioritisation mechanism (Service Levels), due to MPI’s over-abstraction; their usage could significantly improve the performance of prioritisation (offloading into hardware a policy that now is implemented in software).
Chapter 6

Conclusion

6.1 Contributions

This thesis described and evaluated the design and implementation of the mechanisms employed for the distribution of FBX, using RDMA/InfiniBand. Here are the major contributions of the present work:

- design and implementation of stub operators, a scalable and generic software module for system decoupling;
- design and implementation of a fast and reliable networking layer on top of MVAPICH MPI;
- design of strategies to deal with multiple query engines and experiments with multicasting;
- evaluation of the implemented system with standard OLTP, OLAP and mixed workloads.

The system proved to behave well in decoupled mode, particularly for high number of clients, when the power of distribution can be exploited completely; the overhead of the network is not negligible, most of all in latency-sensitive workloads and for under-loaded systems, but the results are reasonably good. Moreover, our solution manages to deal well with a mixed workload, providing good isolation between the analytical and the transactional clients.

MPI has been a powerful library, that let us build a complete networking layer very quickly and easily; on the other hand, it has been hard to optimise and analyse – given its ‘black-box’ nature – and it has proved sometimes too rigid in its API and abstracting too much from the underlying channel. In summary, we think it is a good tool for very large and complex systems where maintainability and readability are the main target, but it can be too limiting in our context, where the focus is on performance.
What we have presented here is just the first step into building a fully distributed rack-scale data processing system. Although some key features are still incomplete or missing, if we look back at the totally non-decoupled system we started from, we can appreciate retrospectively its progress towards a full-fledged distributed system.

6.2 Future Work

The future of FBX is very full of interesting enhancements and analysis directions. The list we present here is just a subset of the opportunities that such a project offers:

- integrate multicast, implement per-machine filtering and perform scalability tests in terms of number of query engines;

- implement data partitioning across machines and perform scalability tests, varying the dataset size and the number of machines of the storage engine;

- implement a query compiler and refactor the workload-specific code, in order to make its maintainability and extension simpler;

- add new workloads from different fields, such as Machine Learning and Graph Processing, and analyse their performance in our batch-based distributed system;

- improve the performance of the networking layer, by developing a library that substitutes MPI;

- design a resource manager and query optimiser, that deals dynamically with core and machine allocation, message prioritisation, broadcasting strategies, query-to-backend matching, exploiting both precomputed performance models and run-time measurements.
Bibliography

[1] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DECOUPLING OF DATA PROCESSING SYSTEMS OVER ROMA/INFINIBAND

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Alessandro

First name(s):

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

17/8/15

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.