University of Waterloo
Faculty of Mathematics

SQL Joins with Interleaved Tables: A Natural Extension of NewSQL Databases

Cockroach Labs
New York, New York, USA

Prepared by
Richard Wu
2B Computer Science
December 2017

Memorandum

To: Vivek Menezes

From: Richard Wu

Date: December 16, 2017

RE: Work Report: SQL Joins with Interleaved Tables: A Natural Extension of NewSQL Databases

As we discussed earlier, I have enclosed the technical report *SQL Joins with Interleaved Tables: A Natural Extension of NewSQL Databases.* for my 2B work report and for the distributed SQL execution engine team at *Cockroach Labs*. This is the third of four work reports that I must successfully draft and complete as part of my BCS Co-op degree requirements mandated by the Co-operative Education Program.

The distributed SQL execution engine (DistSQL) team, for which you are one of the engineering managers, works on the distributed batch processing framework that underlies *CockroachDB's* SQL layer. My role as a Software Engineering Intern was to implement outstanding features outlined in project manifestos on the company's issues board for the DistSQL project. Furthermore, I designed and drafted an RFC for improved SQL joins with interleaved tables, a special variant of SQL tables that *CockroachDB* provides. This report discusses the evolution of large-scale and distributed data applications and how interleaved tables are a natural extension of the DistSQL framework. In addition, the implementation of "interleaved table joins" in *CockroachDB* I was responsible for are highlighted.

The Faculty of Mathematics requests that you evaluate this report for coverage and precision of the technical content and analysis. Following your assessment of this report, a performance evaluation of my work will also need to be completed. The two evaluations will be used to determine whether I receive credit for my co-op term.

I appreciate your assistance in preparing this report.

Richard Wu

Table of Contents

List of Figures

Executive Summary

This report first introduces the business and technical motivations of distributed data applications. The motivations underpin the genesis of *CockroachDB*, a distributed SQL database that provides distributed Structured Query Language (SQL) transaction. The properties of the distributed batch processing framework (DistSQL) and interleaved tables—a variant of SQL tables provided by *CockroachDB*—are introduced. Finally, a more efficient implementation of SQL joins specific to interleaved tables is highlighted.

Distributed databases need to be tolerant to Byzantine failures and serve an important role in addressing an organization's need for a scalable solution to organizing data. *CockroachDB* is one of few horizontally-scalable databases that has SQL semantics.

The distributed batch processing framework in *CockroachDB* transforms a logical SQL plan into physical processors and streams that carry out data operations on the individual servers that house the data. Data is organized as industry-standard SQL tables. An extension of distributed SQL tables are interleaved tables, which improves performance when executing certain SQL queries.

The interleaved SQL table variant in theory permits more efficient SQL joins due to data locality. An initial implementation of improved SQL joins with interleaved tables are **up to 74.3% more performant** than SQL joins between regular, non-interleaved tables.

1.0 Introduction

1.1 The History of Data Applications

The popularization of the Structured Query Language (SQL) and the evolution of relational databases did not come to fruition until after many iterations of previous, ineffective data database models. Back in the 1960s and 1970s, large businesses that handled an enormous amount of data transitioned from the traditional pen and paper form of bookkeeping to processing data on what we know today as mainframes. This was of course when IBM formed as a corporation and became one of the forefathers of modern computing. There was huge demand from businesses for more and more effective ways to organize and retrieve data from computers, a still very foreign concept at the time.

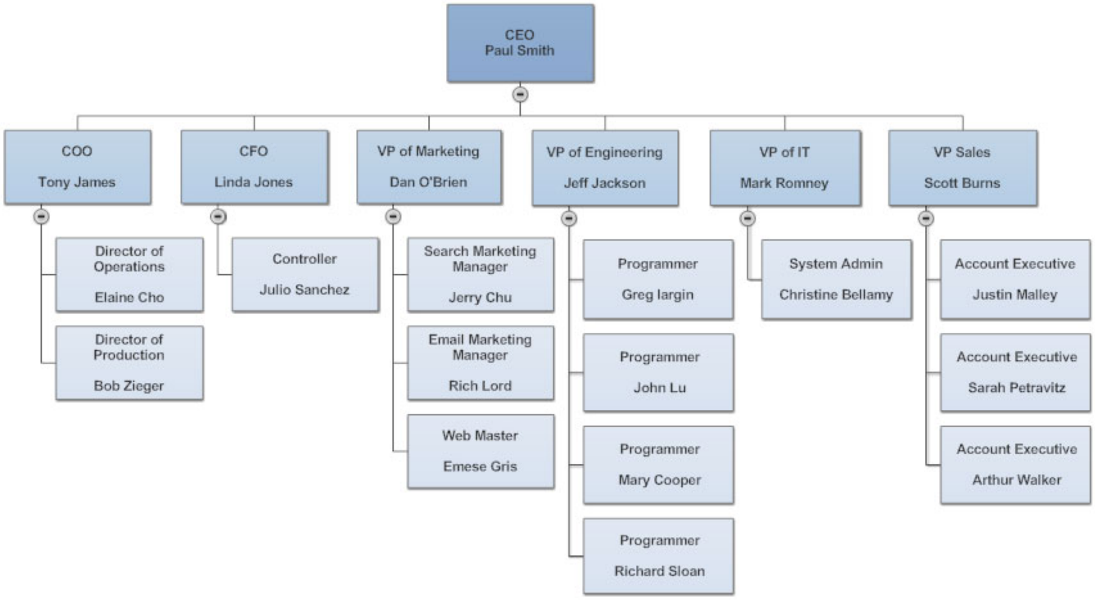
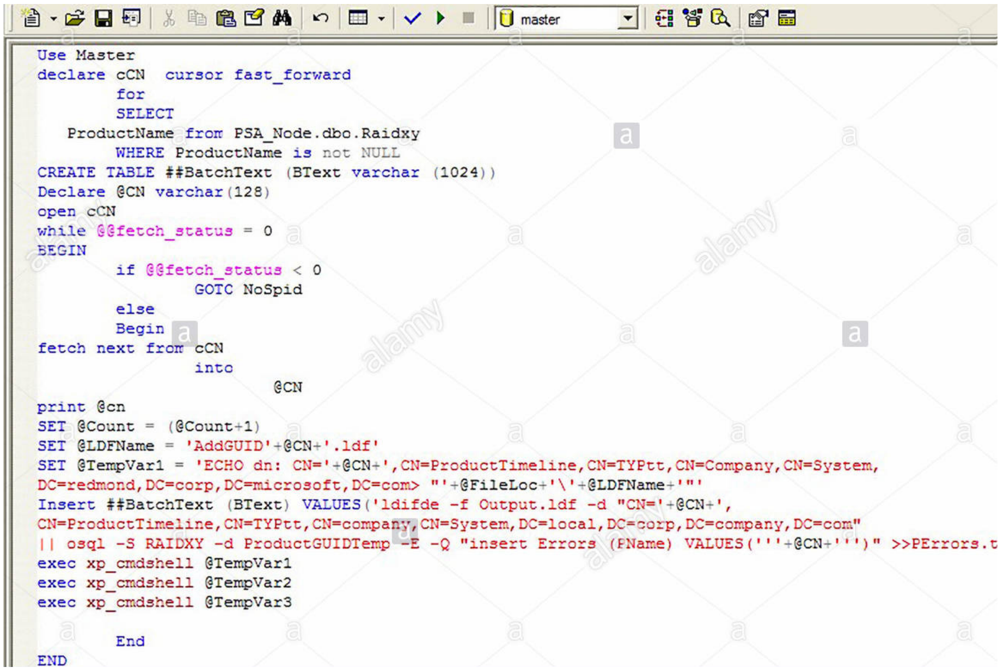


Figure 1 – A typical organization chart of a company. An org chart is a great example of how data is organized in a hierarchy.

Cue hierarchal modelling: a way of organizing data in hierarchies or “levels”. The most familiar hierarchies in our everyday lives is a company’s organizational chart: the staggered tree of executives, managers, and employees. If employee A reports to manager B, then the two have a relationship with each other in the hierarchy.

This kind of modelling can be extended to an unbounded number of applications and is the basis for IBM’s *Information Management System (IMS),* the most popular database for business data processing in the 1970s (Long, Harrington, Hain and Nicholls, 2000). There were many problems uncovered with hierarchal modelling over the years: one such problem is fitting in many-to-many relationships (where an employee C may have multiple managers and the respective managers have multiple employees). You can copy the employee C’s data under the different managers but that introduced the complexity of keeping employee C’s data updated between the various copies (keeping multiple copies of the “same” data is a process called “denormalization”).

Network models became mainstream for an epoch in time which mapped data similar to hierarchal models but addressed the above problem as well as a subset of other problems with hierarchal models. These models were known as *CODASYL* models because they were standardized by the *Conference on Data Systems Language (CODASYL)* committee (Knowles and Bell, 1984). However, both these models had one especially tragic flaw: in order to access the record of data for an arbitrary node in the model (in our example, an employee of the organizational chart), one would have to traverse from the root node (the CEO John Smith) all the way down to desired record. This required knowledge of which branch or "access path" would lead to the correct record and was a huge burden on developers managing these systems.



```
Use Master
declare cCN  cursor fast_forward
        for
        SELECT
   ProductName from PSA_Node.dbo.Raidxy
        WHERE ProductName is not NULL
CREATE TABLE ##BatchText (BText varchar (1024))
Declare @CN varchar(128)
open cCN
while @@fetch_status = 0
BEGIN
        if @@fetch_status < 0
                GOTC NoSpid
        else
        Begin
fetch next from cCN
                into
                        @CN
print @cn
SET @Count = (@Count+1)
SET @LDFName = 'AddGUID'+@CN+'.ldf'
SET @TempVar1 = 'ECHO dn: CN='+@CN+',CN=ProductTimeline,CN=TYPtt,CN=Company,CN=System,
DC=redmond,DC=corp,DC=microsoft,DC=com> "'+@FileLoc+'\'+@LDFName+'"'
Insert ##BatchText (BText) VALUES('ldifde -f Output.ldf -d "CN='+@CN+',
CN=ProductTimeline,CN=TYPtt,CN=company,CN=System,DC=local,DC=corp,DC=company,DC=com"
|| osql -S RAIDXY -d ProductGUIDTemp -E -Q "insert Errors (PName) VALUES('''+@CN+''')" >>PErrors.t
exec xp_cmdshell @TempVar1
exec xp_cmdshell @TempVar2
exec xp_cmdshell @TempVar3

        End
END
```

Figure 2 – A code snippet of *Microsoft's* SQL implementation. SQL (read: relational databases) were originally described in Edgar Codd's 1970 report on *A Relational Model of Data for Large Shared Data Banks*.

In 1970 Edgar Codd, a researcher at *IBM*, published *A Relational Model of Data for Large Shared Data Banks* that set the stage for the largest revolution of information retrieval theory the industry has seen to date (Codd, 1970). The brief yet insightful 11-page report introduced the concept of "relational models" built on the rigorous foundations of relation algebra. In short, relational models map business queries down to a few simple operations:

selection (filtering data), projection (selecting only certain parts of data), and set operations like unions of set (to combine or disassociate multiple sets of data) to name a few. This model of organizing and retrieving data was then mapped to actual implementations of databases and was later formalized as the *Structured Query Language (SQL) standard*. This form of data model has stood the test of time and is one of if not the most popular data querying languages in the world.

1.2 SQL at Scale: *CockroachDB*

One of the finer nuances of the SQL standard is the notion of transactions. Much like the financial transactions that occur when we purchase groceries from the store or order and have them delivered through *Amazon*, a transaction guarantees that a set of actions occur together and without overlapping with other transactions.[1] For example, if my bank account had $25 in it and I tried to buy $25 worth of goods from each of two separate merchants, my bank would decline the second transaction. In the context of a database managing both the bank account of the merchants and me, the system would realize that an update in the balance of my bank account is occurring in two transactions and only permit one to fully execute (or “commit”) in order to maintain the invariant of a non-negative account balance.

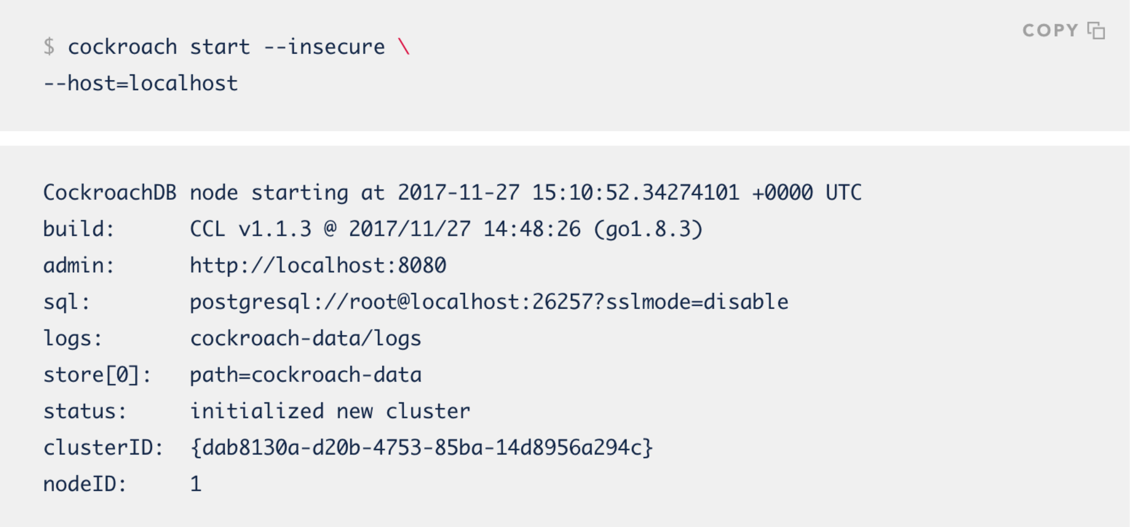
[1] This is a rather partial and informal definition of SQL transactions: the more complete picture of transactions is encapsulated as ACID-compliance.

There are several proprietary and open-sourced SQL databases that are tried and true when it comes to handling production workloads. *Oracle SQL* and *Microsoft SQL Server* are some of the largest proprietary Relational Database Management Systems (RDBMS) whereas *Postgres* and *MySQL* are some the front-runners on the open-sourced side. Note that these are all “management systems”: the value provided is the software and software license, although companies such *Oracle* offers some specialized hardware that work “better” with their system.

Until the last decade or two, most businesses would vertically scale their databases by running larger and more powerful machines in order to handle increasing number of requests. As the industry matured, companies that simply out-scaled the largest possible machine had to look for new ways to handle the exponential growth in user traffic. *Google*, the enormous tech giant that has some of the highest number of daily-active users in the world across all their products, initially started using *MySQL* as the primary RDBMS for their advertising backend (Corbett, Dean, Epstein et al., 2012). Once they could no longer handle the sheer traffic with just one instance of *MySQL* running on one machine, they had to “shard” their *MySQL* data onto multiple machines. This perspective on scaling is referred to as “horizontal scalability”.

Sharding is a common way to deal with scaling out a database that is designed to run on single server machines: at a high level, the data is partitioned into multiple chunks and the chunks each live on separate instances of *MySQL* (or whichever RDBMS) running on individual machines. This form of sharding is implemented in the application's code which is a significant complexity burden on developers. Furthermore, transaction semantics discussed previously will need to be built by the developers outside the scope of the RDBMS. There are also a myriad of other inherent architectural problems sharding introduces that needless to say: very few if none have managed to perfect application-level sharding, including *Google*.

In *Google's* 2012 seminal paper on *Spanner: Google's Globally-Distributed Database*, the researchers and engineers reveal to the industry their implementation of a database that handles all the complexities of sharding, replication (for machine failures), and transaction guarantees in a distributed environment (Corbett, Dean, Epstein et al., 2012). On top of Spanner, there is also *Google F1,* which is a layer built on top of Spanner that permits distributed SQL queries and joins. Together they are a robust answer to an ACID-transactional SQL database (coined as "NewSQL") that scales as a linear function of the hardware provided.



```
$ cockroach start --insecure \
--host=localhost
```

```
CockroachDB node starting at 2017-11-27 15:10:52.34274101 +0000 UTC
build:      CCL v1.1.3 @ 2017/11/27 14:48:26 (go1.8.3)
admin:      http://localhost:8080
sql:        postgresql://root@localhost:26257?sslmode=disable
logs:       cockroach-data/logs
store[0]:   path=cockroach-data
status:     initialized new cluster
clusterID:  {dab8130a-d20b-4753-85ba-14d8956a294c}
nodeID:     1
```

Figure 3 – One-line deployment of a *CockroachDB* instance with command output. Introducing additional nodes to the cluster also consists of one-line shell commands.

While *Google* does offer a cloud-hosted version of *Spanner* and *F1* (*Cloud Spanner*), many companies that wish to have full control over their customer's data by keeping the data in their own datacenters have very few options. *CockroachDB* is an open-sourced distributed NewSQL database that is heavily influenced by *Spanner* and *F1* and supports the Postgres dialect of SQL out-of-the-box. Since distributed databases are inherently cumbersome, *CockroachDB* puts heavy emphasis on ease-of-deployment: it is encapsulated in an all-in-one executable binary whereby a cluster across multiple machines can be started with a few simple shell commands.

2.0 Analysis

The remainder of this report and the primary content in the *Analysis* section will highlight the distributed batch processing framework (the DistSQL engine) that allows SQL queries to be efficiently parallelized onto the numerous machines that store individual “shards” of the relevant data. An inherent property of the DistSQL engine is improved performance from data locality at a cluster level: pieces of data that live together on the same machine can be processed immediately. This is especially relevant for SQL joins, a fundamental aspect of all SQL-compliant RDBMSes.

2.1 Distributed Execution Engine

A SQL query can be broken down into a logical plan: an abstraction of the query that breaks it down into its fundamental components. These components include the relational algebra operations discussed earlier (e.g. selection and projection) as well as other semantics introduced over the years both in the SQL standard and in specific SQL implementations.

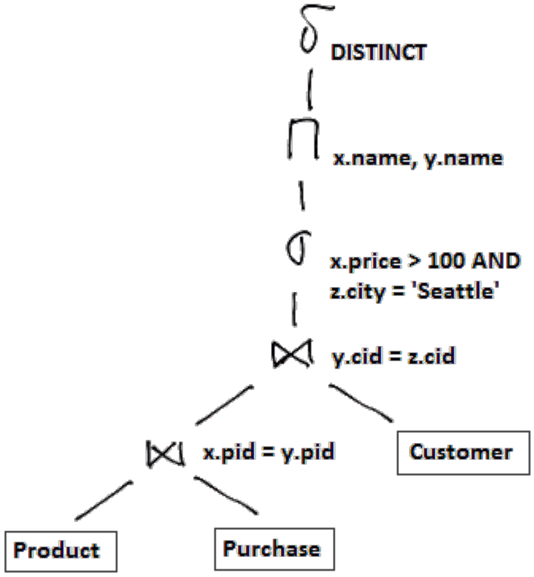


Figure 4 – Representation of a logical plan that is derived from a SQL query. Logical plans in *CockroachDB* resemble n-ary tree structures with each operation represented as a node in the tree.

The logical plan very much resembles a tree structure with nodes as logical operations and edges as input-output relationships. The logical plan then needs to be mapped to actual structures in the code that physically iterate through the data and perform their respective operations, whether it be filtering for SQL rows that have a certain value in a field or selecting a subset of fields from each row. *CockroachDB*'s initial implementation of an execution engine is modelled after the Volcano model (Graefe, 1990). The Volcano query processing model takes the logical plan almost as-is and defines a few methods on each node: namely *Next()*. What *Next()* does is retrieve the next logical row of the operation at that node. The node would need to retrieve rows from its children nodes by invoking *Next()* on them. This propagates all the way down to the leaf nodes and eventually those nodes pull data from the actual disk or SSD drives. The rows are then propagated back up like how lava erupts from inside a volcano.

This form of query execution is simple to imagine on a single node, but gets more complicated when data could be flowing from multiple servers storing individual shards of the relevant data. One could pipe all relevant rows onto one node and apply the Volcano model, but this is rather inefficient especially if many rows are filtered out or only a subset of fields are used. It would be more performant to perform as many operations on the data nodes before sending the intermediary rows to the final node for final processing.

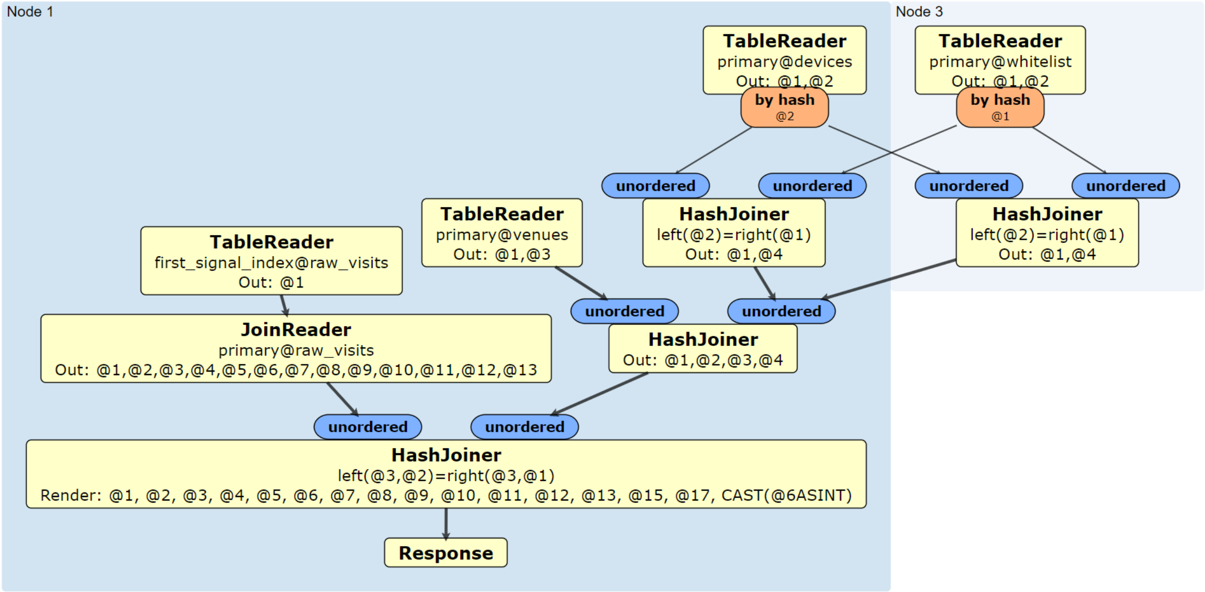
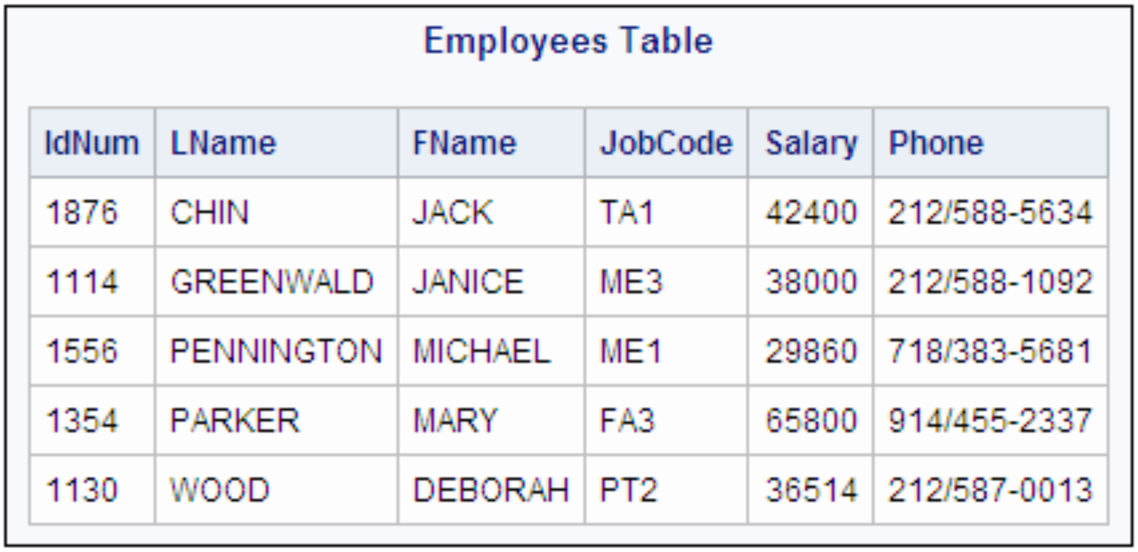


Figure 5 – A distributed execution plan in *CockroachDB*. Each individual logical plan node is transformed or mapped to physical processors. These processors iterate through the data and performs certain operations like joining rows from two tables. Some of the operations can be performed on the node containing the data (e.g. node 3 in the diagram). Intermediary rows are sent back to the final node (node 1) for final processing.

This is the philosophy of the distributed execution engine (colloquially DistSQL) whereby the logical SQL plan is transformed into a dataflow of processors, routers, and streams.

2.2 Interleaved Tables



Employees Table

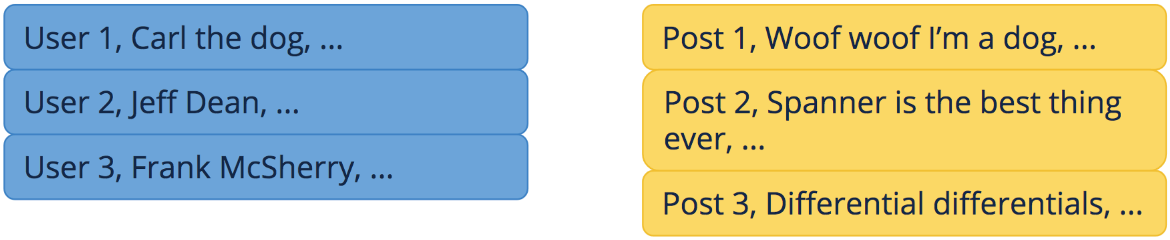
IdNum	LName	FName	JobCode	Salary	Phone
1876	CHIN	JACK	TA1	42400	212/588-5634
1114	GREENWALD	JANICE	ME3	38000	212/588-1092
1556	PENNINGTON	MICHAEL	ME1	29860	718/383-5681
1354	PARKER	MARY	FA3	65800	914/455-2337
1130	WOOD	DEBORAH	PT2	36514	212/587-0013

Figure 6 – A SQL table for all employees of a company. Each row corresponds to one record of one employee. The columns contain fields relevant to each employee such as phone numbers and salary figures.

In SQL, there is the notion of tables and rows: a table contains a collection of rows that all have the same columns or fields. In the context of sharding, many databases arbitrary break apart and re-balance data across the machines specified in the cluster. For example, *Apache Cassandra*, a NoSQL[2] distributed database that operates on tables with rows and columns much like SQL tables, arbitrarily partitions data in the cluster into individual parts or “ranges” (DataStax, 2014). Through an efficient load balancing algorithm called “consistent hashing”, the ranges are assigned to nodes in the cluster. It is important to

[2] NoSQL is a slight misnomer: *Cassandra* does offer a SQL-like query grammar. The NoSQL categorization namely refers to the fact that it does not support distributed ACID transactions.

note that a given table occupy a range or adjacent ranges such that even in the event of re-balancing, the rows for the table tend to stay together on a node or a couple of the same nodes. This partitioning scheme is similar to the partitioning policy *CockroachDB* employs to distribute data by breaking apart tables into chunks called “ranges”.



User 1, Carl the dog, ...

User 2, Jeff Dean, ...

User 3, Frank McSherry, ...

Post 1, Woof woof I’m a dog, ...

Post 2, Spanner is the best thing ever, ...

Post 3, Differential differentials, ...

Figure 7 – Two SQL tables, Users (left) and Posts (right), where each row corresponds to one user or blog post, respectively. In this example, blog Post 1 belongs to User 1 and similarly for Post 2 & 3. The two tables have a one-to-one relationship.

The core philosophy of SQL and relational databases is “normalizing” data as much as possible: keeping only one copy of a related set of data in self-contained tables. For example, if we wanted to store data for a blog where we have users who make posts, we would like to separate the user data into one table called *Users* and the blog post data in another table called *Posts*. Whenever we want to find the corresponding blog posts for a given set of users, we can perform an SQL join. Similarly, if users also had comments stored in a table *Comments*, we can join *Users* with *Comments* to retrieve corresponding comments for a set of users.

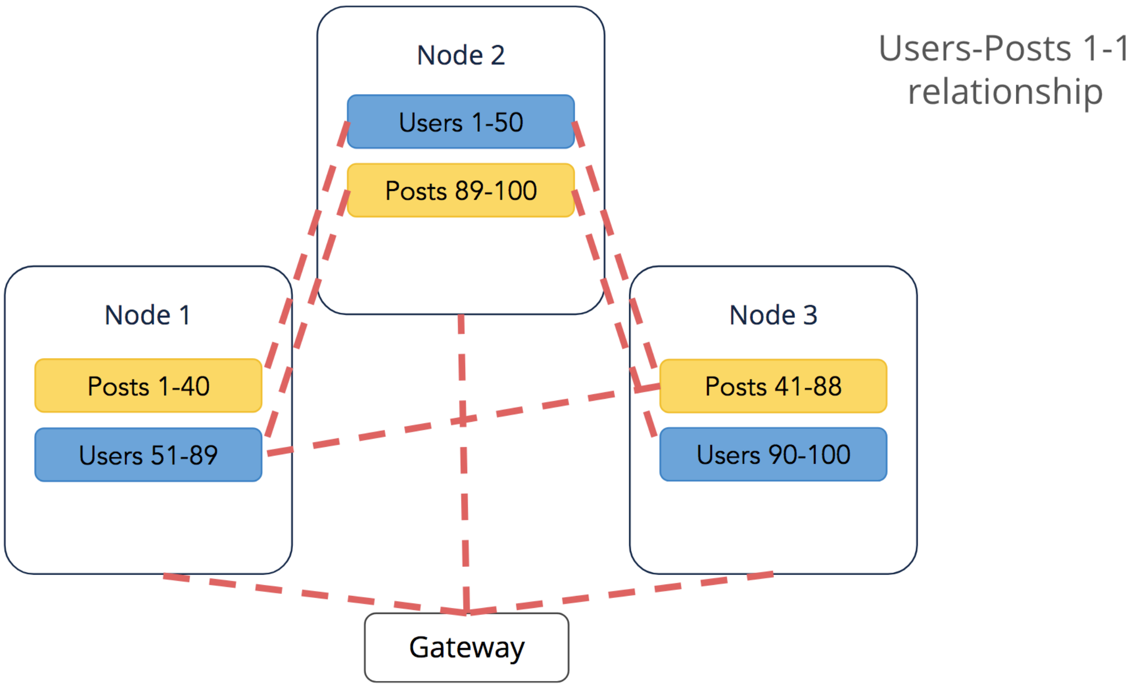


Figure 8 – An example of how *Users* and *Posts* tables can be partitioned and distributed across three nodes. The red-dotted lines indicate the remote procedure calls (RPCs) required to perform a SQL join between blog post entries and their corresponding user entries. Since RPCs are relatively slow, we'd like to reduce the RPC traffic as much as possible for a given SQL query.

Since the sharding layer of *CockroachDB* is agnostic of SQL tables (instead a SQL table occupies a contiguous section of the underlying storage as described before) *Users* and *Posts* may be unfavorably partitioned such that many inter-node Remote Procedure Calls (RPCs) are required to match up corresponding rows from both tables (see Figure 8). While an individual table is grouped together with high data locality (e.g. Users 1 to 50 are grouped together in one range on *node 2* in Figure 8), there is no such guarantee *between* multiple tables.

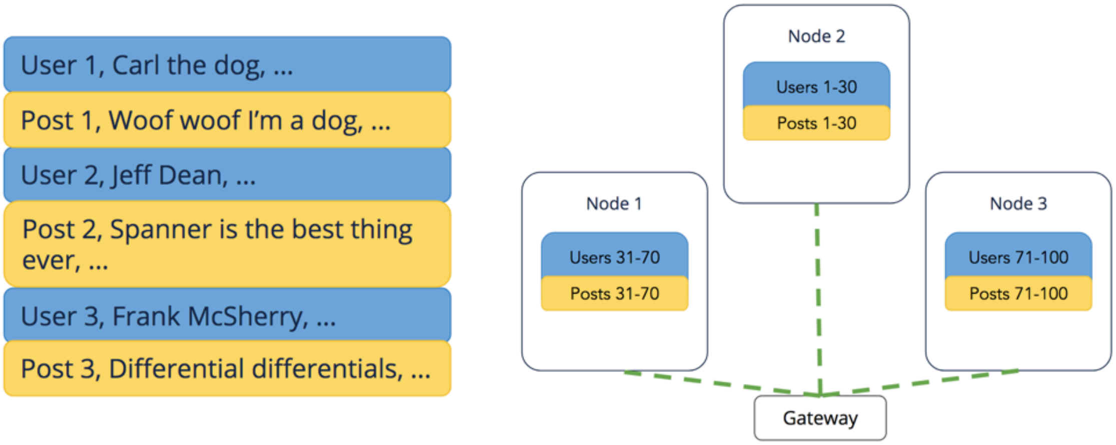
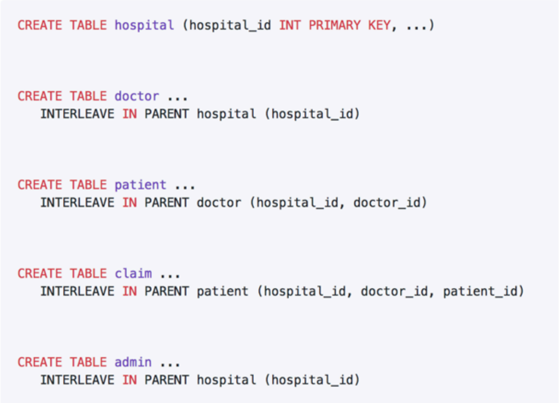

User 1, Carl the dog, ...
Post 1, Woof woof I'm a dog, ...
User 2, Jeff Dean, ...
Post 2, Spanner is the best thing ever, ...
User 3, Frank McSherry, ...
Post 3, Differential differentials, ...
Node 2
Users 1-30
Posts 1-30
Node 1
Users 31-70
Posts 31-70
Node 3
Users 71-100
Posts 71-100
Gateway

Figure 9 – The same *Users* and *Posts* table in *CockroachDB* but with *Posts* interleaved into *Users*. Effectively, *Posts* and *Users* are grouped together such that they conceptually form one table. *CockroachDB* partitions and distributes the two tables such that SQL joins between corresponding entries require few RPCs (green-dotted lines) relative to non-interleaved tables in Figure 8.

Interleaving tables introduce the idea of storing rows from some table after a row (or some rows) of another table (see Figure 9). Both *CockroachDB* and *Cloud Spanner* permit Database Admins (DBAs) to create interleaved tables to improve data locality and performance for SQL operations such as joins. *Oracle* has something similar called "multi-table cluster indexes" (Burleson, n/d). The join logic for interleaved tables will need to be slightly modified to take advantage of the convenient lockstep-like (read: interleaved) arrangement of rows between two interleaved tables.

2.3 Interleaved Table Joins

In the example between *Users* and *Posts* in Figure 9, joining rows from the two interleaved tables is rather trivial: every row of *Posts* is nested immediately after its corresponding row in *Users.* However, this becomes more complicated once you have multiple tables interleaved into each other.



```
CREATE TABLE hospital (hospital_id INT PRIMARY KEY, ...)


CREATE TABLE doctor ...
   INTERLEAVE IN PARENT hospital (hospital_id)


CREATE TABLE patient ...
   INTERLEAVE IN PARENT doctor (hospital_id, doctor_id)


CREATE TABLE claim ...
   INTERLEAVE IN PARENT patient (hospital_id, doctor_id, patient_id)


CREATE TABLE admin ...
   INTERLEAVE IN PARENT hospital (hospital_id)
```

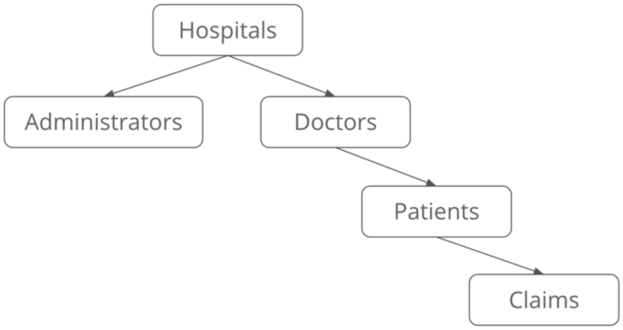


Figure 10 – (Left) how interleaved tables are declared and created in *CockroachDB*. (Right) The interleaved tables represented in its tree form. Note that there is a strong resemblance between this tree structure and hierarchies in hierarchal modelling: this is because interleaved tables form an “interleaved hierarchy” (in graph theory, an interleaved hierarchy is an “arborescence”).

Interleaved tables work exceptionally well in practice with data that naturally forms a hierarchy and are often queried together. In traditional SQL databases like *Postgres* or *MySQL*, one would have to perform several joins (multi-table joins) to match up data from multiple related tables. In the example from Figure 10, someone who wants to retrieve all *Doctors*, *Patients,* and *Claims* for some *Hospitals* would have to perform a 4-way join. This can become rather costly in a distributed database without interleaving as demonstrated with the simple two-table case in Figure 8.

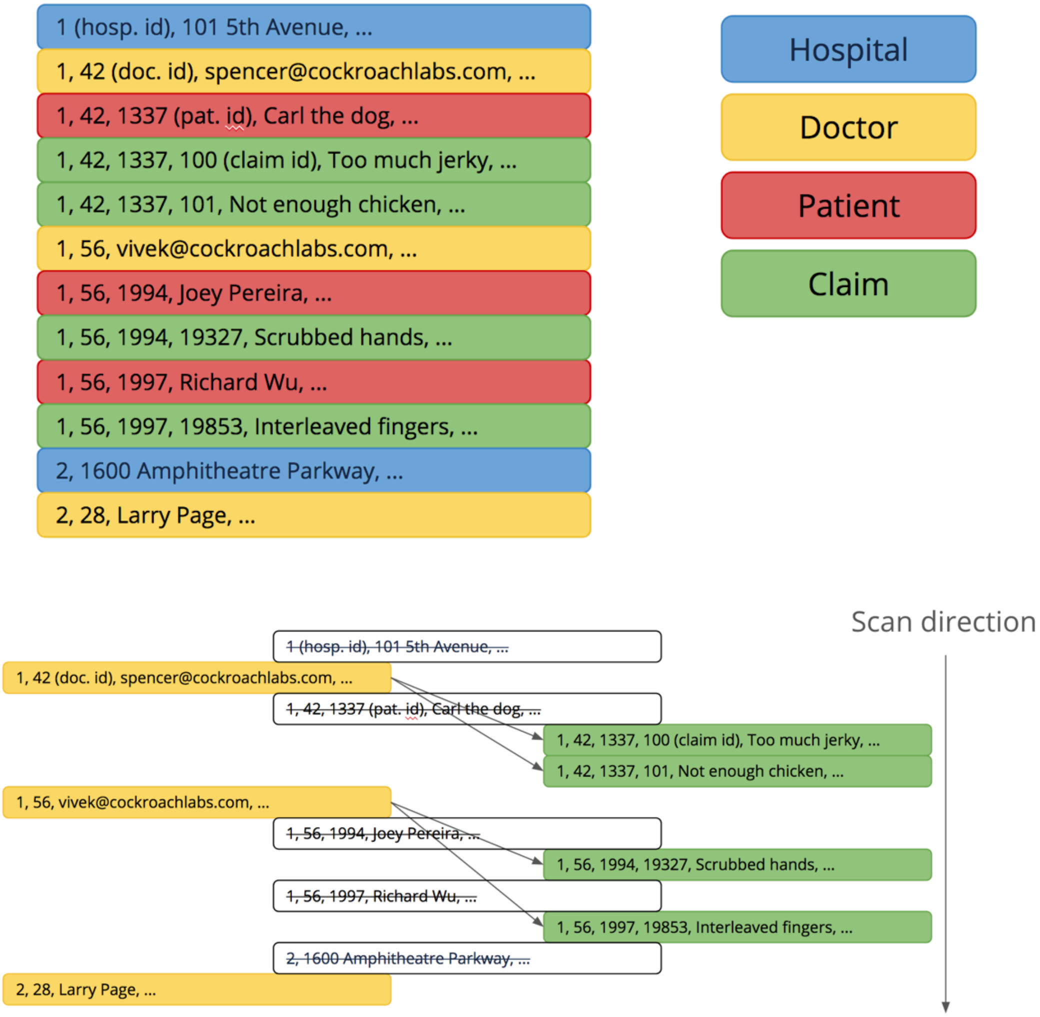

1 (hosp. id), 101 5th Avenue, ...
1, 42 (doc. id), spencer@cockroachlabs.com, ...
1, 42, 1337 (pat. id), Carl the dog, ...
1, 42, 1337, 100 (claim id), Too much jerky, ...
1, 42, 1337, 101, Not enough chicken, ...
1, 56, vivek@cockroachlabs.com, ...
1, 56, 1994, Joey Pereira, ...
1, 56, 1994, 19327, Scrubbed hands, ...
1, 56, 1997, Richard Wu, ...
1, 56, 1997, 19853, Interleaved fingers, ...
2, 1600 Amphitheatre Parkway, ...
2, 28, Larry Page, ...
Hospital
Doctor
Patient
Claim
Scan direction
~~1 (hosp. id), 101 5th Avenue, ...~~
1, 42 (doc. id), spencer@cockroachlabs.com, ...
~~1, 42, 1337 (pat. id), Carl the dog, ...~~
1, 42, 1337, 100 (claim id), Too much jerky, ...
1, 42, 1337, 101, Not enough chicken, ...
1, 56, vivek@cockroachlabs.com, ...
~~1, 56, 1994, Joey Pereira, ...~~
1, 56, 1994, 19327, Scrubbed hands, ...
~~1, 56, 1997, Richard Wu, ...~~
1, 56, 1997, 19853, Interleaved fingers, ...
~~2, 1600 Amphitheatre Parkway, ...~~
2, 28, Larry Page, ...

Figure 11 – (Top) Example of how rows from the interleaved tables in the example from Figure 10 may be stored together. (Bottom) A visualization of how a query that wants to join rows between Doctors and Claims is conceptually executed.

In Figure 11, a set of data with the same schema as the example from Figure 10 is shown with an abstract representation of how a join query between rows from *Doctors* and *Claims* is executed with interleaved table joins. Since interleaved tables from an arborescence, there are certain invariants we can apply to create a more efficient join algorithm than naïve implementations of nested loop, hash and/or merge joins. The narrowness of this margin doesn't permit me to include the precise details[3], but the full Request For Comments (RFC) for interleaved table joins (courtesy of yours truly) can be accessed in the publicly-available repository.[4]

2.4 Performance of Interleaved Table Joins

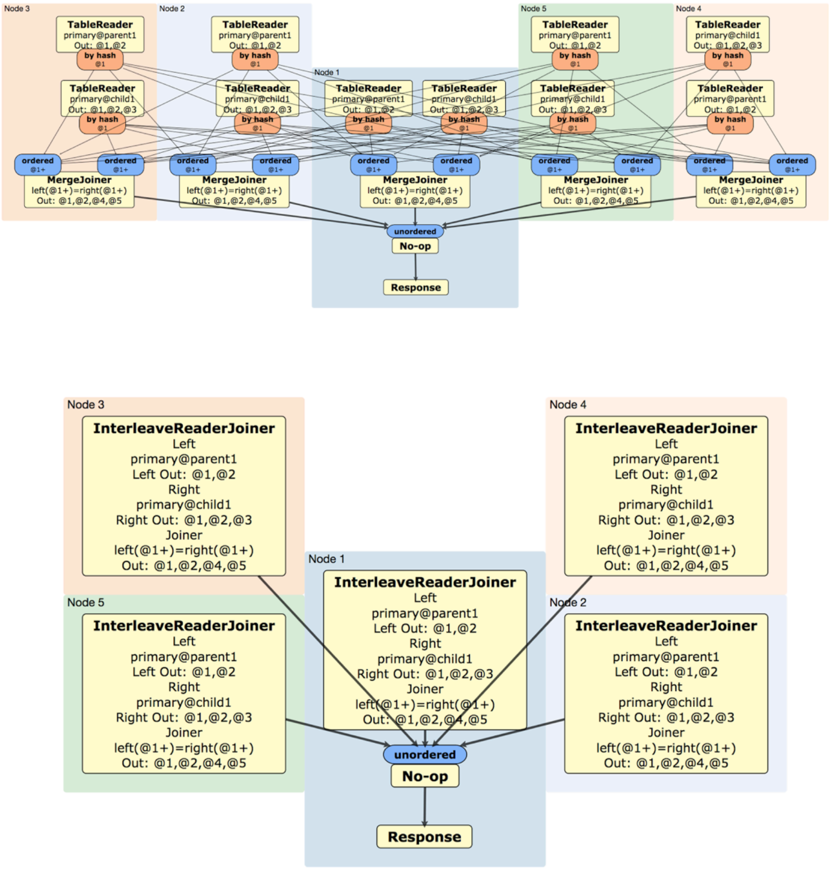


Figure 12 – DistSQL execution plans for before and after comparison. (Top) Naive merge joins between two interleaved tables. (Bottom) More efficient interleaved table joins between the same two interleaved tables. It is obvious that there is a drastic reduction in RPC traffic in the latter which clearly manifests itself in the performance numbers.

[3] https://en.wikipedia.org/wiki/Fermat's_Last_Theorem

[4] https://github.com/cockroachdb/cockroach/blob/master/docs/RFCS/20171025_interleaved_table_joins.md

An important part of any systems software development is measuring how an implementation improves the product or feature. In this case, it is important to compare how the implementation of interleaved table joins compare to joins between non-interleaved tables and naïve joins (i.e. joins that are agnostic of the interleaving property) between interleaved tables. From intuition, we hypothesis and expect that our improved implementation of joins in the context of interleaved tables should meet the following criteria:

1. Interleaved table joins should be **comparable** to joins on non-interleaved tables in slightly pessimistic cases.
2. Interleaved table joins should be **strictly better** than naïve joins for interleaved tables.
3. Interleaved table joins should perform **1.5x – 2.0x** better than regular tables in ideal use cases (i.e. joins with highly-hierarchal data).

A benchmark[5] was written to exercise join queries on non-interleaved and interleaved tables in *CockroachDB* both before and after the implementation of interleaved table joins. Several different scenarios were proposed and benchmarked to establish lower and upper bounds across a variety of settings.

[5] https://github.com/cockroachdb/loadgen#interleave

Scenario	Non-interleaved tables		Interleaved tables (naïve)		Interleaved tables (improved)	
	QPS	99th %tile latency (ms)	QPS	99th %tile latency (ms)	QPS	99th %tile latency (ms)
Simple 2 tables 1 range	5.5	1610	4.0	2282	5.3	1745
Pessimistic 4 tables >1 ranges	7.55	1779	0.4	17450	0.9	8590
Typical 4 tables >1 ranges	1.35	6443	1.1	8724	2.0	4429
Ideal 2 tables >1 ranges	3.5	2684	3.1	3423	6.1	1712

Scenario	Vs non-interleaved (% change)		Vs naïve interleaved (% change)	
	QPS	99th %tile latency (ms)	QPS	99th %tile latency (ms)
Simple	-3.6%	+8.4%	+32.5%	-23.5%
Pessimistic	-88%	+382.9%	+125%	-50.8%
Typical	+48%	-31.3%	+81.8%	-49.2%
Ideal	+74.3%	-36.2%	+96.8%	-50%

Figure 13 – Tables summarizing throughput (in queries per second or QPS) and tail latency performance numbers across non-interleaved tables, interleaved tables with naive joins, and interleaved tables with the improved implementation. Equivalent queries were concurrently executed against the 3-node *CockroachDB* cluster by 8 workers on a machine with 4 cores.

From our performance experiments[6], we see that our initial hypotheses hold true. That is: the new implementation is strictly better than a naïve approach to joining between

[6] https://github.com/cockroachdb/cockroach/issues/20586

interleaved tables. In typical and ideal cases where data forms a natural hierarchy, interleaved tables with the improved join logic outperforms non-interleaved tables.

3.0 Conclusions

The history of how large-scale data applications transitioned from hierarchal modelling to relational modelling to a hybrid of both is important to understanding where the latest innovation in information retrieval. In some sense, the industry has come full circle back to hierarchal modelling where data locality is important in the context of sharding. An example of a hybrid approach that takes aspects from the 1960s hierarchal models of data and from the time-tested relational movement are interleaved tables. Interleaved tables in *CockroachDB* as well as in other distributed NewSQL databases like *Cloud Spanner* can offer greater performance for certain topologies of data by grouping data together when data is distributed across many nodes. They also permit some subset of relational operations that much more efficient than a pure hierarchal model.

References

Donald K. Burleson: “Reduce I/O with Oracle Cluster Tables,” dba-oracle.com.

Edgar F. Codd: “A Relational Model of Data for Large Shared Data Banks,” Communications
of the ACM, volume 13, number 6, pages 377–387, June 1970. doi:
10.1145/362384.362685

James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: “Spanner: Google’s Globally-Distributed Database,” at 10th USENIX Symposium on Operating System Design and Implementation (OSDI), October 2012.

“Apache Cassandra 2.0 Documentation,” DataStax, Inc., 2014.

Graefe, G. (1990). Encapsulation of parallelism in the Volcano query processing system (Vol. 19, No. 2, pp. 102-111). ACM.

J. S. Knowles and D. M. R. Bell: “The CODASYL Model,” in Databases—Role and Structure: An Advanced Course, edited by P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, pages 19–56, Cambridge University Press, 1984. ISBN: 978-0-521-25430-4

Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls: IMS Primer. IBM Redbook SG24-5352-00, IBM International Technical Support Organization, January 2000.

Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., ... & Cieslewicz, J. (2013). F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, *6*(11), 1068-1079.