No Relation: The Mixed Blessings of Non-Relational Databases

by

Ian Thomas Varley, M.S.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2009

The Report committee for Ian Thomas Varley

Certifies that this is the approved version of the following report:

No Relation: The Mixed Blessings of Non-Relational Databases

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____
Adnan Aziz

Co-Supervisor: _____
Daniel Miranker

Dedication

To my wife Jill,

without whose unending support

I most certainly would not be here today.

Acknowledgements

I would like to acknowledge the generous support of my many professors at the University of Texas who have given graciously of their time in support of my education and research over the past two years; especially, Professors Daniel Miranker and Adnan Aziz, who advised on this project; and Professors Christine Julien and Joydeep Ghosh, whose courses and research heavily informed the work herein.

14 August 2009

Abstract

No Relation: The Mixed Blessings of Non-Relational Databases

Ian Thomas Varley, M.S.E.

The University of Texas at Austin, 2009

Co-Supervisors: Adnan Aziz and Daniel Miranker

This paper investigates a new class of database systems loosely referred to as "non-relational databases," which offer a subset of traditional relational database functionality, in exchange for improved scalability, performance, and / or simplicity. We explore the differences in conceptual modeling techniques, and examine both the advantages and limitations of several classes of currently available systems, using running examples of real-world problems as implemented in both a traditional relational database model, as well as several non-relational models.

Table of Contents

List of Figures

SECTION 1: INTRODUCTION

The history of the relational database has been one of continual adversity: initially, many claimed that mathematical set-based models could never be the basis for efficient database implementations; later, aspiring object oriented databases claimed they would remove the "middle man" of relational databases from the OO design and persistence process. In all of these cases, through a combination of sound concepts, elegant implementation, and general applicability, relational databases have become and remained the lingua franca of data storage and manipulation.

Most recently, a new contender has arisen to challenge the supremacy of relational databases. Referred to generally as "non-relational databases" (among other names), this class of storage engine seeks to break down the rigidity of the relational model, in exchange for leaner models that can perform and scale at higher levels, using various models (including key / value pairs, sharded arrays, and document-oriented approaches) which can be created and read efficiently as the basic unit of data storage. Primarily, these new technologies have arisen in situations where traditional relational database systems would be extremely challenging to scale to the degree needed for global systems (for example, at companies such as Google, Yahoo, Amazon, LinkedIn, etc., which regularly collect, store and analyze massive data sets with extremely high transactional throughput and low latency). As of this writing, there exist dozens of variants of this new model, each with different capabilities and trade-offs, but all with the general property that traditional relational design—as practiced on RDBMS systems like Oracle, Sybase, etc.—is neither possible nor desired.

The aim of this paper is to explore the conceptual design space of non-relational databases as compared to traditional relational databases. It is clear that the design needs

of the two paradigms are different, but how fundamental are the differences, and what strategies can we use to transition our conceptual designs from one to the other?

In Section 2, we introduce a **running example**, with some in-depth analysis of the problem scenarios and their solutions, first in relational SQL database designs, and then in some example non-relational database designs. This will introduce the basic concepts of the non-relational database concepts in an informal way, and begin to lay out the groundwork for the detailed explorations that follow.

In Section 3, the **benefits** of the key/value store approach will be explained in depth, in terms of simplicity (fewer services lead to less complexity), scalability (weaker integrity assumptions lead to more dimensions of concurrency), and raw performance (fewer features means fewer layers to pass through).

Section 4 further explores the **detriments** of moving from a relational database to a non-relational database, specifically related to impoverished modeling constructs and consistency guarantees: the effects of denormalization, lack of relational integrity, lowered expressive power, and potential lack of ACID properties.

Following the “good cop / bad cop” discussion of sections 3 and 4, Section 5 will provide a detailed **survey** of many of the currently available non-relational database store implementations, comparing several dimensions of features and / or modeling concepts that each of these systems employ.

Section 6 then introduces several **design strategies** that might guide our thinking about conceptual design and its transition into the non-relational world. Some suggestions are made about key/value modeling conventions that retain some of the advantages of relational databases, as well as design patterns for methodically transforming one to the other.

Finally, Section 7 provides **analysis and conclusions,** offering a vision for a future path that database technologies can tread to attempt to gain benefits from both paradigms.

Note: the focus of this paper is on the *conceptual* data design options available within a non-relational store as compared to traditional relational database design. It does not directly deal with issues of performance, scalability, cluster distribution and management, etc., except insofar as touching on these topics is required to understand the rationale behind the core concepts of non-relational stores. The topics of performance and scalability alone would far outstrip the scope of this report, especially considering how widely they vary across the implementations we have surveyed. There are convincing arguments to be made regarding the scalability and performance advantages gained from non-relational stores, in the right situations, which justify their emergence and continued development. Interested readers are encouraged to delve into the **Bibliography** section to find more references on these topics, or more importantly, to engage in their own research efforts to understand the performance characteristics of these systems in the context of their own work.

NOTES ON DIAGRAM STYLE

This paper uses a slightly restricted dialect of UML for describing the logical and physical schemas of traditional relational database designs, based in part on the modeling conventions of [Hay, 1995]. It differs from standard UML in the following minor ways:

- Rather than using a single descriptor on relationships, which can be ambiguous regarding the directionality of the relationship, we typically use two role names at

the ends of the relationship, indicating the nature of the relationship as it would be used in a sentence. This allows us to translate directly from diagrams into sensible English, such as "Every Employee works for one Company; a Company may employ many Employees."

- The traditional annotations "1", "0...1", and "0...*" are used to indicate the cardinality of a relationship. For additional clarity on the multiplicity of relationships, crows' feet are also used to indicate the "many" side of a relationship, as this aids in quick visual interpretation of data diagrams. These annotations (as with role names) are retained in the transition from logical to physical diagrams, though in the latter they do not have any special properties beyond documentation. If the labels represented many-to-many relationships in the logical model, the same names are retained and used only once in the Physical model, because the junction table is only used as a physical implementation, not a logical design.
- By convention, the direction of crows' feet always points up and to the left on diagrams (with the exception of "many to many" relationships on logical diagrams, which obviously have crows' feet in both directions). This has the effect of placing "concrete" entities towards the bottom/right side of the diagram, and derived or relational entities towards the top left, and generally establishes a standard "flow" to diagrams, making them easier to interpret quickly.
- Navigability arrows are never included, as data entities are typically considered directionless and have navigability in both directions in all cases.
- Aggregation / composition indicators (diamonds) are not used, mainly because the information they add is not an inherent part of modeling the physical representations of the examples used in this report in today's relational databases.

- Entities begin with a capital letter, and attributes begin with a small letter.
- The third section of an entity diagram, which is used in object modeling to show operations but in data modeling to show keys and relationships, is only displayed when this information would not be redundant and / or obvious. Typically, the "PK" and "FK" markers next to attributes are sufficient to prevent ambiguity. “PK” markers next to multiple attributes indicate a composite primary key; "PFK" indicates an attribute that is both a primary and foreign key.

NOTES ON TERMINOLOGY

In this report, the words *DBMS* and *database* are used interchangeably. This is contrary to the prescriptive usage, which says that *database* should always refer to the actual collection of data, whereas *DBMS* (or “Data Base Management System”) should always refer to the software which manages the collection of data (the same goes for the RDBMS, or Relational DBMS; and the NRDBMS, or Non-Relational DBMS).

There is nothing wrong with this prescriptive usage; however, the common descriptive usage of these terms is that they are interchangeable and can be understood based on context. If we refer to the *capabilities* of a database, we are clearly speaking of DBMS software, because raw data has no capabilities per say. If we refer to some entities or attributes *contained in* a database, we are clearly speaking of it as a collection of data.

SECTION 2: INTRODUCTION BY EXAMPLE

We will begin our exploration of the differences in conceptual modeling for relational and non-relational databases using a simple example that grows more complicated over time.

ONE TABLE: JOB OPENINGS

Consider the following simple scenario: a business wants to advertise job opportunities on their website. Given a set of open positions maintained by the HR department, with a handful of attributes for each, we want to display this information dynamically on a public-facing web page.

This scenario essentially describes what every database (relational or otherwise) would refer to as an "entity" or "table". We will label this entity as "Position", and give it several sample attributes. We represent this using a simple UML diagram:

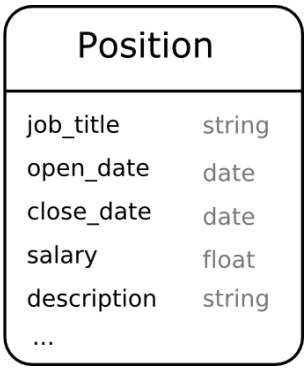

Figure 1: UML diagram of a single entity, Position

The organization might advertise several Positions, with job titles such as "Accountant" and "Night Janitor", each with attributes such as "salary", "description", "location", etc. Each would be a single tuple in this simple relation.

To use a relational database to power this information[1], we define a physical relational model for it, which looks similar to the logical model in this case. The only difference is the addition of a *primary key*, which uniquely identifies each position. Common industry practice in relational databases is to use auto-incrementing integer fields as primary keys for many entities, rather than to construct complex primary keys that reflect particular (and possibly misunderstood) business rules. In this case, we have added an "id" attribute to Position as its primary key (because, for example, there might legitimately be multiple positions with the same title, start date, end date, etc.).

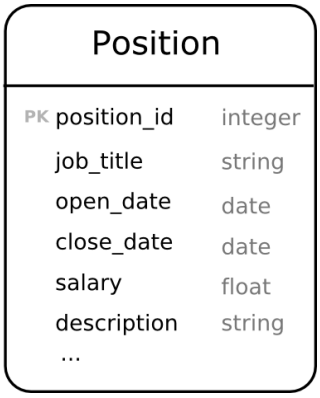

Figure 2: Physical Relational Database Model for a Single Entity

Our simple application now consists of merely reading and writing records in this table. Regardless of the technology used to implement our database—be it an RDBMS, a

[1] Of course, we have little impetus to use a full relational database for such a simple example; we could just as easily write the information in a flat file or XML document; but bear with us, as the example will get more complex.

non-relational key/value store, or a flat file—our conceptual model is identical: one entity.

MANY TO ONE: BASIC EMPLOYMENT APPLICATION

Having seen the ease with which we completed this request, our HR department has now come to us with a new task: they would like to allow potential future employees to fill out their personal information via the web page, and apply for jobs online. Further, they would like the ability to do queries across all applicants, to help narrow the search for the perfect person for the job; for example, "Show me all applicants in New Jersey who have experience as an electrician and are willing to relocate ...", etc.

Logical Model

Consider the most basic addition to our logical schema: there are now Applicants, each of which is related to one Position:

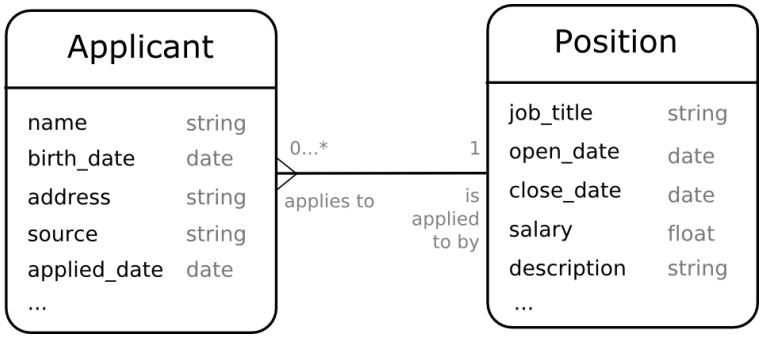

Figure 3: Logical model for many-to-one relationship

Individuals would choose a Position (e.g. "Accountant"), and fill in their personal details, creating Applicant records for any position they are interested in[2]. This is the classic *many-to-one* relationship in data modeling; one Position is related to any number of Applicants, and each Applicant is related to only one Position.

Relational Physical Model

Moving into a relational database physical schema, we can adorn the logical diagram with several new attributes that act as primary and foreign keys for relational database tables:

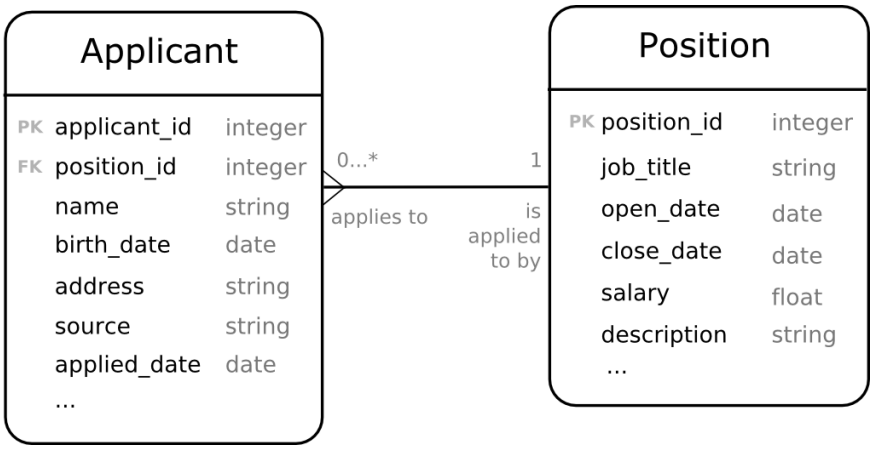

Figure 4: Physical model diagram for a Many-to-one relationship

As with our Position table, we have added an identifier field to indicate the uniqueness of each instance of an Applicant (since, for example, one person might

[2] There are naturally many other facets of the real-world situation that could be included here, such as the fact that one applicant might realistically apply for multiple jobs, in which case we could, say, give them a user account and password with which to manage their multiple applications. We'll ignore that level of detail for now, in favor of the simpler many-to-one model.

theoretically apply multiple times, even to the same position)[3]. We then connect the two entities via a foreign key relationship on the "position_id" attribute of the Applicant entity, indicating that each Applicant "applies to" exactly one Position, and each Position may be "applied to by" multiple Applicants.

Sample data for this physical table layout might be something like[4]:

```
Table "Position":

position_id job_title  open_date  close_date  salary    description
----------- ---------- ---------- ----------- --------- -----------
1001        Accountant 4/1/2010   5/1/2010    50000.00  Cooks the books
1002        Janitor    6/1/2010   7/1/2010    30000.00  Cleans the loo
…

Table "Applicant":

applicant_id position_id name           birth_date  state       applied_date  …
------------ ----------- -------------- ----------- ----------- ------------ -
30001        1001        Ned Flanders   4/5/1958    Nevada      9/1/2009      …
30002        1001        Homer Simpson  7/1/1962    Texas       10/2/2009     …
30003        1002        Bill Smith     1/1/1900    California  11/4/2009     …
…
```

This relational model is quite straightforward: there are only two entities, connected by a single relationship, and in the basic case, this enables the entire range of functionality described in the problem statement. We can craft a simple SQL query to show us only the open positions:

```
SELECT *
  FROM
    Position
  WHERE
    open_date <= CURRENT_TIMESTAMP()
    AND close_date >= CURRENT_TIMESTAMP()
```

[3] Again, we could have used a composite primary key, which would be the more pure approach in set mathematics, but the practice of assigning a primary key id field is nearly ubiquitous in commercial application development

[4] Note that the "..." indicate both additional rows in the relation, as well as additional attributes, like "phone number", "years of experience", etc.

We can use INSERT and UPDATE statements to create and modify the information for a specific applicant:

```
INSERT INTO Applicant (
    position_id,
    name,
    birth_date,
    ...
) VALUES (
    @position_id,
    @name,
    @birth_date,
    ...
)
```

We can then query against a join of these two tables to see a full report of all applicants and the positions they applied to. We can also restrict this search by giving WHERE clauses against any of the attributes in either table, such as a query for all applicants in New Jersey who are applying for jobs with salaries of over $100,000, sorted by name:

```
SELECT P.job_title, A.name, A.birth_date, ...
  FROM
    Position P
    INNER JOIN Applicant A
        ON A.position_id = P.position_id
  WHERE
    P.salary > 100000
    AND A.state = 'New Jersey'
  ORDER BY
    A.name
```

Our use of a relational database completely hides the specific implementations used to achieve these ends - finding relevant job postings on disk and caching them in memory, writing new applicant records to disk, merging the information about positions and applicants in memory, filtering the results by Boolean expressions, sorting the results, etc; the declarative nature of SQL syntax completely isolates us from these

details. Aside from the possibility of speeding up future searches by creating indexes (which may be desirable for performance, but is not required for correctness), we are finished with the entire specification of the data definition and access, and can immediately write additional business logic on top of this framework to enforce business rules, display forms, etc.

Non-Relational Model

How would we recreate this simple data model design under a non-relational schema? As an example, we will describe an implementation using the Google App Engine data store, since its syntax in Python is simple and clear, and it was specifically created to be simple and reminiscent of relational databases, while only providing the services typical of key/value stores because of its implementation as a massively scalable cloud computing service.

The two entities of our logical data model, Position and Applicant, become the two data objects, or Entities, in our non-relational data model:

```
class Position(db.Model):
    job_title = db.StringProperty(multiline=False)
    open_date = db.DateTimeProperty(auto_now_add=False)
    close_date = db.DateTimeProperty(auto_now_add=False)
    salary = db.StringProperty(multiline=False)
    description = db.StringProperty(multiline=True)
    ...

class Applicant(db.Model):
    position = db.ReferenceProperty(Position)
    name = db.StringProperty(multiline=False)
    birth_date = db.DateTimeProperty(auto_now_add=False)
    address = db.StringProperty(multiline=False)
    source = db.StringProperty(multiline=False,
       choices=set(["employee referral", "recruiter", "advertisement"]))
    applied_date = db.DateTimeProperty(auto_now_add=True)
    ...
```

Each of these classes, the Position and the Applicant, can be thought of as its own distributed hash table; every tuple has a key (system-assigned in this case) which is used as the hash locator value, and a "value" which is all the other information about the record. There are exactly 3 operations that can be done on the data store hash table: put, get, and delete. Beyond that, the database engine itself offers few additional features.

Notice first that we have actually moved back in the direction of our original logical model; there are no "id" properties on these entities, because each instance of an entity is automatically given a system-designated "key" property which is its key into the data storage engine. A shadow of relational integrity can be intimated by using keys from one entity as properties of another, as the following code snippet illustrates:

```
pos = Position()
pos.job_title = "Accountant"
pos.put()

app = Applicant()
app.position = pos.key()
app.name = "Homer Simpson"
app.put()
```

We will further explore the implications of this degree of relational integrity below.

Getting a list of the currently active positions implies using a filtered query, which is supported by the App Engine when we create an index that covers the fields in question[5]:

```
positions = Position.all()
positions.filter("open_date <", date.now).filter("close_date >", date.now)
for position in positions:
    # display the position in the list ...
```

[5] As mentioned above, there are restrictions on this filtering ability in that the results ultimately need to appear in a single index in contiguous order, and thus cannot use arbitrarily complex inequality comparison operators on multiple items

Getting a list that is a "join" of Applicants with their Positions, however, is a harder task. To maintain our sort order (ascending by Applicant name), we must first iterate over Applicants, and then for each Applicant, we must retrieve the data about what position it was for:

```
applicants = Applicant.all()
applicants.filter("state =", "New Jersey")
for applicant in applicants:
    position = applicant.Position()
    # show data containing attributes of both the position and applicant
objects
```

Notice that here, for the first time, we are doing a fair amount of work in the client tier that was done for us automatically in the relational model, with a JOIN operation. The work in this example is not complex, but for arbitrarily complex multi-way joins, this could get quite confusing and error prone[6]. The subject of doing efficient in-memory joins for large database tables is a heavily studied and optimized area of research, and for the biggest cases, it is highly unlikely that a developer of average skill would correctly implement the level of sophistication in, for example, a two-phase multi-way merge sort, or a hash join.

Consider also that if our application offers multiple sort orders as a feature (for example, by clicking on the column headers in a grid to re-sort), we might need to either cache the intermediate result in memory, or construct multiple versions of the code that construct and sort the values in different ways. Caching the values in memory is not difficult, but might not be possible for very large data sets; the relational database properly abstracted the situation for us in either case, but the non-relational database does not.

[6] This begs the question as to why our data is structured in such as way as to even require large multi-way joins, if we are not using a relational database paradigm; this is a question we will return to later.

It should be clear by this point that there is some (potentially large) class of operations that we can achieve declaratively, with no effort, in a SQL database, which require significant programming in a non-relational database. That said, there is no evidence of a lack of expressiveness; everything we were able to do with our relational schema, we have been able to faithfully mimic with the non-relational schema, albeit with some addition of effort for the case of more complex queries. Let us next move on to an extension of this example that gives the non-relational database the upper hand.

MANY-TO-MANY: QUESTIONS AND POSITIONS

Consider now that our Human Resources department has returned to us and suggested that each open position might actually need a different set of questions - that is, instead of just one standard set of questions regardless of position, we now need to ask different questions depending on the job, and render the form dynamically, changing continually as users imagine new and ever more exciting questions for future employees. We might ask the accountant to declare what year he or she got a CPA, whereas we might ask the Night Janitor to list "years of mopping experience". Of course, all the other requirements -- the need to create open positions, get applicant input, and enable searches and reporting on the resulting applicant pool -- are still in effect.

Let us further assume, for the sake of example, that there is a requirement that new questions can be added at any time by the administrative users of the system, without developer or DBA input - i.e. without any actual schema changes to the relational database design. We conclude, therefore, that we will now need a *Question* entity, with sufficient information to dynamically display input forms (for example, labeling, type, ordering, etc). While there can be many arrangements and subtleties to this relationship,

let us assume for sake of example that this is a many-to-many relationship, where each question exists only once, but can appear (or not appear) on any number of Positions' forms:

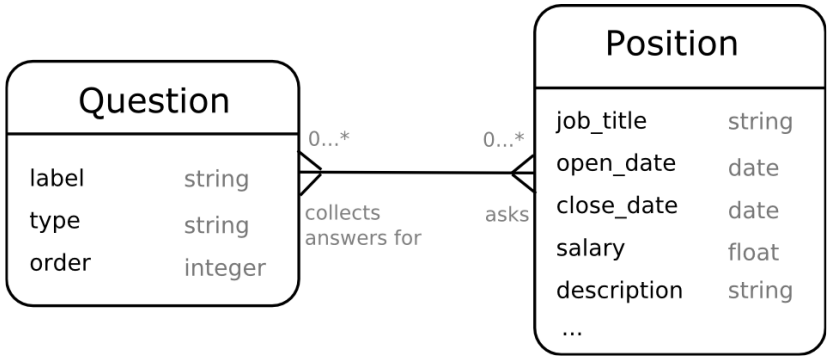

Figure 5: Logical data model for Many-to-many relationship

Bridging into the physical model world, this becomes a three-table relationship, as follows:

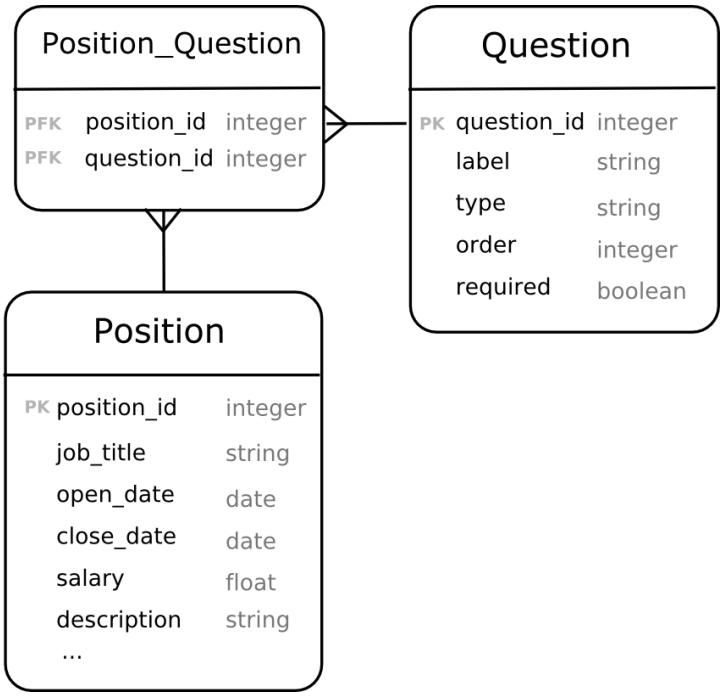

Figure 6: Physical data model for Many-to-many relationship

We can now define any number of Questions, and pick which Questions appear on the form for each Position.[7] Sample data for these questions might be something like:

```
Table "Position":

position_id     job_title   open_date   close_date  salary    description
--------------- ----------- ----------- ----------- --------- -----------------
--
1001            Accountant  4/1/2010    5/1/2010    50000.00  Cooks the books
1002            Janitor     6/1/2010    7/1/2010    30000.00  Cleans the loo
...

Table "Question":

question_id     label           type        order
--------------- --------------- ----------- -------
101             Name            string      1
102             Birth Date      date        2
103             State           string      3
104             CPA Date        date        4
105             Years Mopping   number      5

Table "Position_Question":

position_id     question_id
--------------- ------------
1001            101
1001            102
1001            103
1001            104
1002            101
1002            102
1002            103
1002            105
```

Thus, we have associated the first 3 questions to both positions, and then associated "CPA date" only to the accountant position, and "Years Mopping" only to the janitor position.

[7] depending on the specific business requirements, attributes of the Question class might properly move to the association class - for example, it could be required to have each Question appear in a different order on the form depending on which Position is being shown. We leave these details out for clarity, as that has no impact on the important concepts in this case.

The many-to-many relationship of questions to positions has an interesting design pattern when translated to the non-relational world. Consider an implementation of this logical design in another product type, the family of BigTable systems (which also includes open source implementations such as Hypertable and HBase). In this setup, we have an essential key/value paradigm, but the data storage engine does more with the data in the value itself, providing a more thorough structure and meta-structure. Each entity can have "column families" (of which there are a discrete and limited number, established at design time), and with a column family, there can be an unlimited number of "columns" (which are effectively repeating cells within the column family).

To establish the many-to-many relationship above, we need model only two entities in this paradigm: the Question and the Position, which relate to each other by including a column family to hold instances of the relationship:

Position Row	Column Families	
	Info:	Question:
<position_id>	**Info:title** **Info:open_date** **Info:close_date** **Info:salary** **Info:description**	**Question:<question_id>**

Question Row	Column Families	
	Info:	Position:
<question_id>	**Info:label** **Info:type** **Info:order**	**Position:<position_id>**

Figure 7: BigTable schemas for many-to-many relationship

Now, any position can contain its relationship to any number of questions, and any question can contain its relationship to any number of positions. Because of the repeating nature of columns within a column family, we have broken down the single-value barrier in relational database design that forces us to use an intermediate table to connect entities in this way.

There are, of course, ramifications of this type of design; the same information is represented in two different ways, which could theoretically differ. We will address this concern below under the topics of relational integrity and consistency. For the moment, consider only that we have indeed satisfied our conceptual design using a structure outside of the traditional relational database design paradigm.

ENTITY/ATTRIBUTE/VALUE: EXTENSIBLE APPLICATION FIELDS

While explicitly storing questions seems to be a simple way to satisfy our new requirements, it belies the difficulty we have introduced for ourselves in another area. Things that were formerly the province of the schema itself are now *data* in the schema. In the relational world, this puts us in a bind regarding what to do with the Answers to these questions. We can no longer rely on the Applicant entity having strongly named attributes for each possible question on the form (e.g. "birth date", "address", etc.). With a relational database, we effectively have two choices, which we will refer to as the "unstructured" method, and the "structured" method.

In the unstructured method, we could change the Applicant table to contain arbitrary (unnamed) storage, either in a series of individual fields (aka "buckets")[8]:

[8] We have kept both "applied date" and "source" as permanent, system-supplied fields in this design, for reasons that will become clear below.

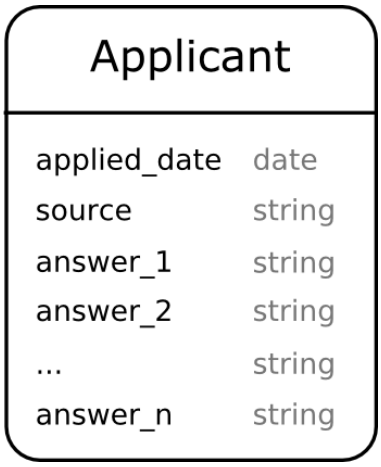

Figure 8: Entity with “bucket” columns

Or alternately, it can be modeled with a single "blob" field:

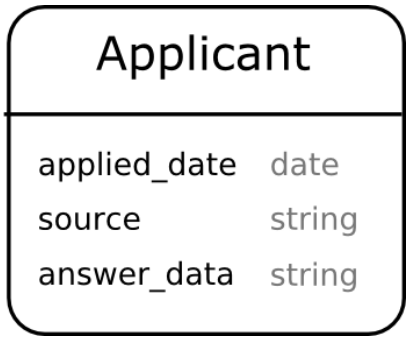

Figure 9: Entity with “blob” column

It is then up to the system's code to enforce rules about putting the right answers into the right buckets, and / or providing a meaningful internal structure to the data in the blob field. SQL provides no intrinsic way of querying data in this form; for example, our earlier query returning the job title, name, and birth date of an applicant becomes more

difficult in the bucket method, requiring us to impute a mapping between the positional column and the question, and thus dooming us to construct the SQL statement dynamically for every query:

```
SELECT P.job_title, answer_1 as 'name', answer_2 as 'birth_date', ...
  FROM
    Position P
    INNER JOIN Applicant A
        ON A.position_id = P.position_id
  WHERE
    P.salary > 100000
    AND A.answer_3 = 'New Jersey'
  ORDER BY
    A.answer_1
```

Further, we have effectively eliminated the benefit of relational integrity here. There is nothing in the database design enforcing the fact that the values that appear in the "answer_1" column are actually names, or that the position being applied for even asked for the Applicant's name. We have effectively relegated the relational database to storing flat, undifferentiated data. It is only slightly better, from a querying point of view, than using the blob method (which is essentially impossible to query, short of using complicated string pattern matching queries against the text blob itself, which are almost sure to perform miserably and be difficult to write in SQL.)

The alternative approach, which would be the more "correct" solution in standard relational database design, is to structure the data with proper normalization, and create a new entity that relates to Applicants in a many-to-one relationship, as follows:

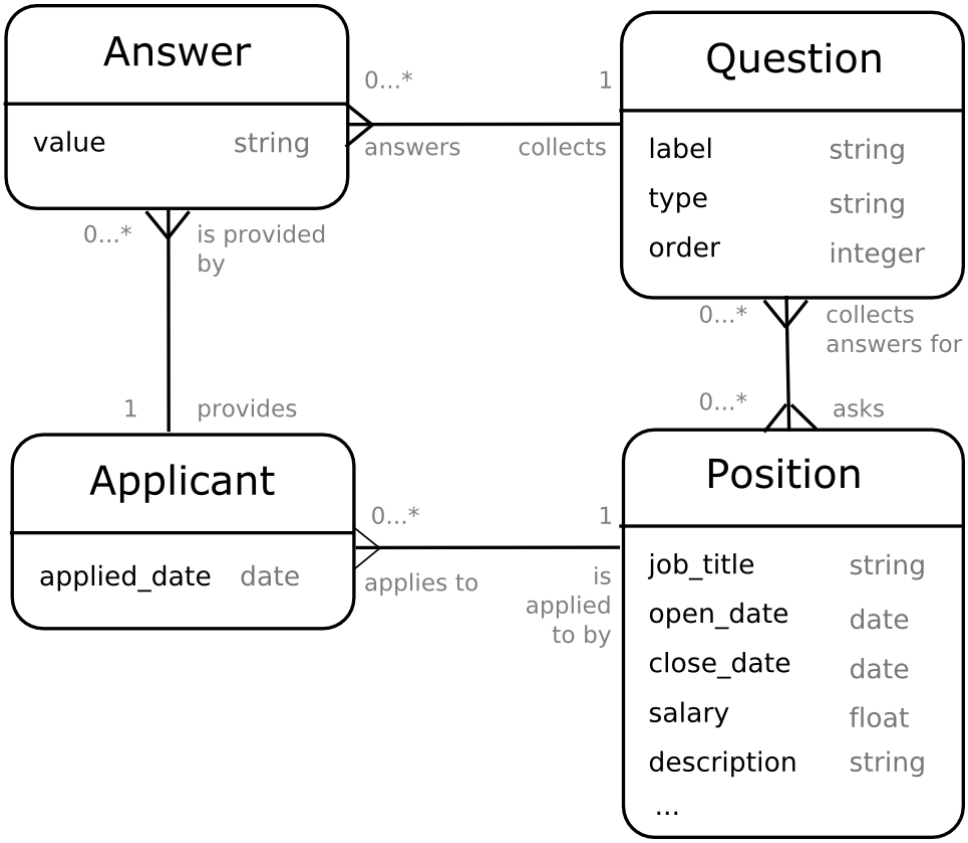

Figure 10: Normalized question / answer model

Our new "Answer" table stores one record per answer, keyed by the applicant ID and question ID[9]. In other words, the attributes that were columns in our original Applicant relation now become rows in this new relation, which represents a single Answer by a single Applicant to a single Question.

This logical model maps directly to a physical model, primarily by adding keys. Incorporating the model of questions to positions, the complete picture of the physical relational model is now:

[9] This model doesn't directly depict the fact that the Position attribute of the Applicant entity must imply a record in the Position_Question table with the same position_id and question_id, but that fact could easily be encoded as a CONSTRAINT in a relational database.

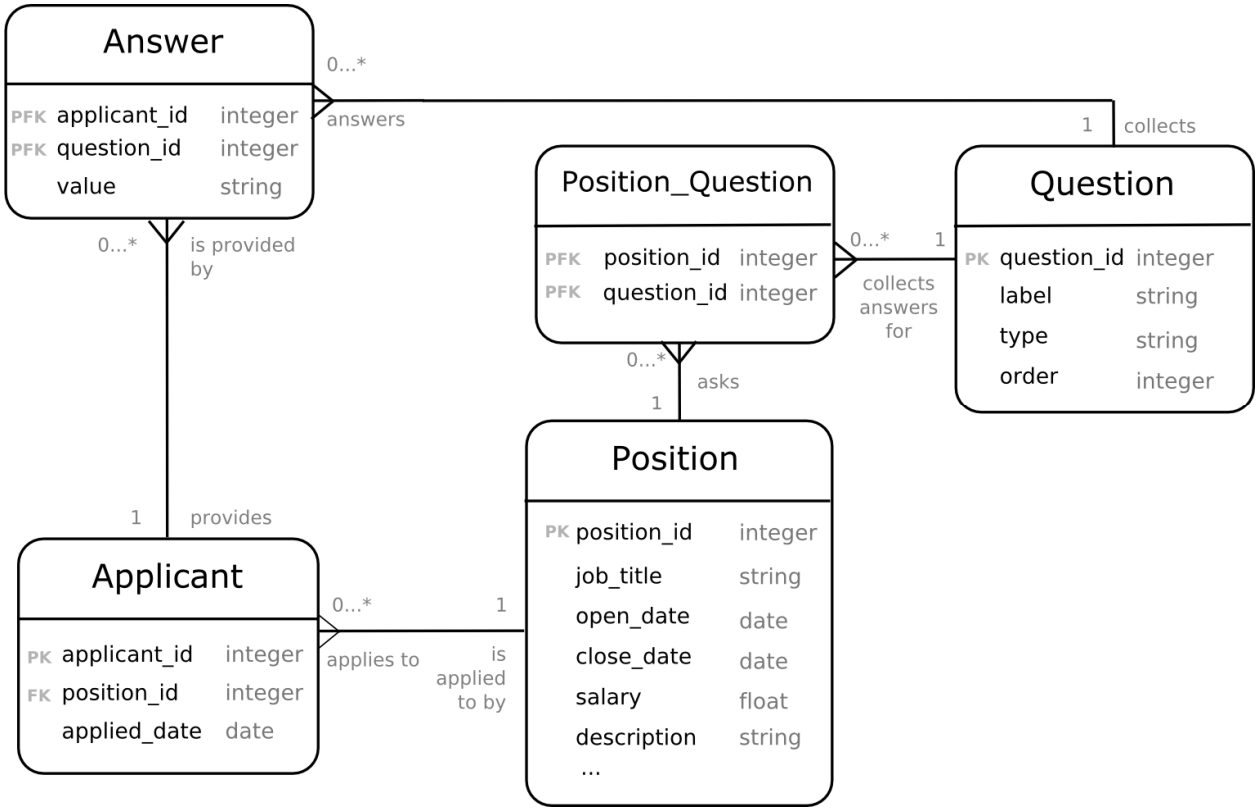

Figure 11: Full physical relational model for questions and answers

This style of data design is a variant of what is sometimes referred to as a "generic" table, or an "open schema" table. It is a common design pattern; other names for it include "EAV/CR", for "Entities, Attributes, Values / Classes, Relationships"; "object-property-value", as used by Object Oriented designers; "frame-slot-value" by the Artificial Intelligence Researchers; and the "Subject-Predicate-Object" triples of Resource Description Format (RDF), which is the basis for the "Semantic Web". Taken to an additional degree of generality, this type of relationship can indeed be used to meta-model any type of data; attributes, and the entities themselves, become facts in a single table that points to an entity identifier (via a key) and an attribute type (via another key).

The sample data from the Applicant table above, transformed into this model, would look like:

```
Table "Applicant":

applicant_id    applied_date  source
--------------- ------------- ------------------
30001           9/1/2009      Employee Referral
30002           10/2/2009     Recruiter
30003           11/4/2009     Employee Referral

Table "Answer":

applicant_id    question_id value
--------------- ----------- ------------------
30001           101         Ned Flanders
30001           102         4/5/1958
30001           103         Nevada
30001           104         recruiter
30002           101         Homer Simpson
30002           102         7/1/1962
30002           103         Texas
30002           104         employee referral
30003           101         Willie Scoggins
30003           102         1/1/1900
30003           103         California
```

Note that for sparse data, this actually turns out to be a very space-efficient representation; each item has some overhead, in terms of its two integer keys, but for many attributes that is typically a small portion of the value (64 bits of key, versus potentially large string attribute values). Storing this data in traditional tabular format, even using only a single byte to represent NULL values (which is unlikely) would end up taking up much more space, assuming a sparse distribution of values.

So, how do we query this data in a way reminiscent of our previous query examples? Not easily! We have successfully modeled our data in a fully normalized fashion ... and in so doing, we have nearly completely crippled our ability to write queries that work with it in a way similar to how we did before. Even a simple tabular result showing the name and birthday of every applicant from New Jersey is *extremely* difficult,

requiring an additional outer join to the answer table for each question we want included - here, "name", "birth date" and "state":

```
SELECT P.job_title, A1.value, A2.value, ...
  FROM
    Position P
    INNER JOIN Applicant A
        ON A.position_id = P.position_id
    LEFT OUTER JOIN Answer A1
        ON A1.applicant_id = A.applicant_id
        AND A1.question_id = @name_question_id
    LEFT OUTER JOIN Answer A2
        ON A2.applicant_id = A.applicant_id
        AND A2.question_id = @birth_date_question_id
    LEFT OUTER JOIN Answer A3
        ON A3.applicant_id = A.applicant_id
        AND A3.question_id = @state_question_id
    ...
  WHERE
    P.salary > 100000
    AND A3.value = 'New Jersey'
  ORDER BY
    A1.value

   >>
```

The same complexity would continue, requiring an additional self-join for each additional attribute; large tabular results are rendered impossible as the query optimizer collapses under the weight of massive join requests.

We have hit upon a situation here where the traditional relational database architecture falls flat[10]. So, how would we achieve this same design goal in a key/value store? This is a case where the inherent design of key/value stores actually lends itself perfectly to our problem. Since the data store ultimately only cares about keys and values, it does not matter if we add additional properties to the value that do not match each other.

[10] Of course, there are ways to mitigate this effect in a relational database, such as using cached or temporary versions of the table that are constructed dynamically and then can be queried normally; there are also a wide range of techniques for automating the extraction and querying from EAV-type designs.

Using the Google App Engine data store's Python API again as an example, we can use the "Expando" class to easily represent a model where properties are added as they are needed. The class itself would simply be modeled as:

```
class Applicant(db.Expando):
    position = db.ReferenceProperty(Position)
    source = db.StringProperty(multiline=False, choices=set(["employee
referral", "recruiter", "advertisement"]))
    applied_date = db.DateTimeProperty(auto_now_add=True)
```

Code to use it would then be along the lines of:

```
janitor = Applicant()
janitor.name = "Montgomery Burns"
janitor.years_of_mopping_experience = 2
janitor.put()

accountant = Applicant()
accountant.name = "Homer Simpson"
accountant.year_obtained_cpa = 1997
accountant.put()
```

The data store has no specific "schema" for these entities in advance, and whatever attributes are assigned are those that are stored. Assuming we are still using a Question table to keep track of all the questions we might want to ask, and some relationship between the Position data and the Question data, then we have done all we need to do in terms of enforcing the integrity of this data set.

How would we query and filter this, as above? For filtering, an index can be built against the data store for any query that might be executed, or manual filtering can be done in the client code based on the values (or even the existence) of properties. This process is not necessarily easier than the process for a relational database, but it is uniform and can be developed on the same level as any other query against the data (which may actually have a beneficial effect on data design overall, a concept we will

explore in detail below). We have put a simple end to all of our worrying about how to represent the schema of this data: we simply don't. Relational databases are not designed for this kind of behavior, and make data designers jump through hoops that ultimately are not even necessary.

In fairness, let it be noted that of course, a relational database can always be made to store anything a key/value store can hold, by defining a simple two column schema with a "key" column and a "value" column that simply holds a binary or text blob. Doing that, however, gets few or none of the gains from non-relational database technology, but incurs all of the losses of it, which we will see in detail below.

ANALYTICAL REPORTING

Imagine now that our favorite HR manager returns with a new request. The "source" attribute of our Applicant entity holds a string indicating where the Applicant heard about the opportunity at the company - for example, an employee referral, a recruiter, Monster.com, etc. The choices for this field might be given in a droplist on the application front end, and / or stated as a constraint on the property itself (as they were in our model above).

Our HR manager is now requesting a "recruiting effectiveness" report, containing information about the efficacy of each possible source of new employees, by number of Applicants. She wants the output to be something like this:

```
source              count
------------------- -------
advertisement       186
employee referral   552
recruiter           415
```

In SQL, this is a simple query using a "GROUP BY" clause:

```
SELECT
    source,
    count(*)
  FROM
    Applicant
  GROUP BY
    source
  ORDER BY
    source
```

No sooner have we understood the problem than we have solved it: this query represents exactly the data in question, returned consistently in real time thanks to the underlying query engine of our relational database.

In our key value store, this is no longer a single "query", but must instead be treated as a manually created collection operation across the entire data store. Each platform has its own specific implementation of this, but the overall idea is well expressed in the "map/reduce" paradigm that originated in functional languages and was popularized by Google [Chang et al, 2006]. In essence, you would write a function that crawled the entire data space, accumulating the values in buckets as needed. You may then cache the result in its own data store, or recalculate it as needed.

This architecture is eminently sensible for the types of problems that key/value stores originated to solve, where the idea of getting a consistent snapshot with transactional consistency of a hugely distributed data store is neither reasonable nor expected. However, this is functionality we have come to expect in SQL, and most relational database designs rely on the ability to express this query simply and execute it efficiently.

Of course, standard SQL is not itself the ultimate panacea for all types of data requests. Using the same example, suppose the hiring manager would like to see this information broken out by year, like:

```
source              2008    2009    2010
------------------- ------- ------- ------
advertisement       85      60      41
employee referral   168     175     209
recruiter           15      80      320
```

Our "SELECT / FROM / WHERE / GROUP BY" pattern can no longer elegantly handle this request, because it involves two levels of grouping: one by column, and the other by row. Only in the OLAP section of the 1999 SQL standard [SQL Standard, 1999] is there an operation that can even produce data in this form:

```
SELECT
    source,
    sum([2008]) as '2008',
    sum([2009]) as '2009',
    sum([2010]) as '2010'
  FROM
    Applicant
      PIVOT (count(*) FOR DatePart(yyyy, applied_date) in ([2008], [2009],
[2010]) A
  GROUP BY
    source
  ORDER BY
    source
```

However, from the key/value store paradigm, this is no more or less difficult than the previous query - it is a simple shift in the calculation, putting the results into a two dimensional matrix instead of a one dimensional vector. Transitioning to higher dimensions, as you might in a data mining effort, for example, is only incrementally more effort; whereas in SQL, is not only more difficult, but completely impossible (short of

using temporary tables, or some other higher level analytical structures such as data cubes, etc).

This distinction points to one of the most important gains in using key/value stores, which we will see in detail below. By restricting the power of the data storage engine to having a much more basic set of primitives, we reduce the tendency to see the world in terms of the set of abilities provided by SQL, and open up the much wider possibilities granted by the map/reduce paradigm. Any Turing-complete language, with all the facilities inherent in full-fledged programming, can be used to generate results. This is a double-edged sword, as we will see.

MASSIVE MULTIPLICITY: KEYWORD SEARCH

Friday afternoon at 4:45pm, the HR manager returns to us with one last urgent request. "When people apply for positions, they usually upload a resume. I want the ability to search against the key terms in these resumes, and find applicants who might have applied for one position but would be a good fit for another. Can you do that?"

How should we answer this request in a relational database? Presumably we can parse through the resumes and pull out lists or sets of words or phrases of interest. We could then create a table for each resume (let us call it a "document" for generality) and then another table that stores one row for each word (or "term") associated with an applicant. The physical and logical models might be:

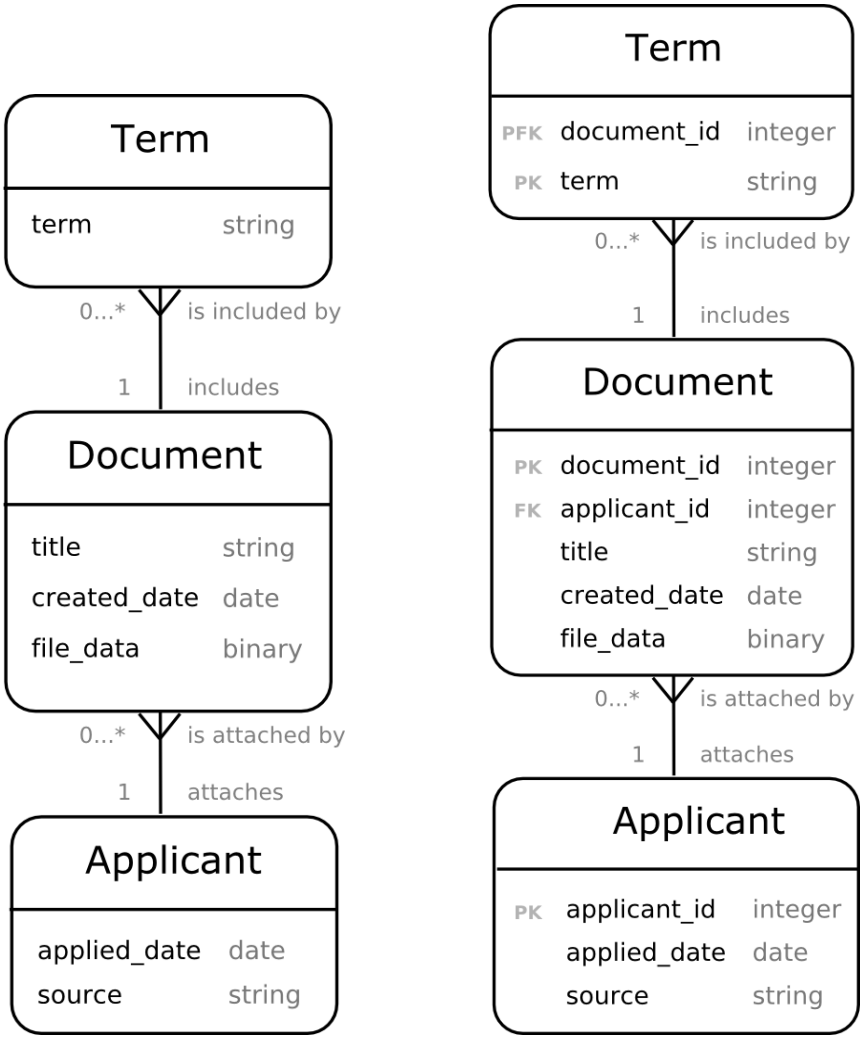

Figure 12: Logical (left) and Physical (right) models of term storage within documents

This design implies several constraints about the manner in which we are collecting and keeping terms: for example, that we only keep each term once per document (implied by the Primary Key) and that there is no explicit relationship between the same word kept in multiple different documents, other than its exact spelling. We could of naturally modify this approach in a variety of ways, by restricting the set of words to only those that we are interested in, or keeping counts of how many times the

word is used, etc. However, for the simplicity of this example, we will presume that this level of detail is sufficient.

At this point, doing a search for all the applicants who have entered a the keyword "CPA" in their resume looks like this:

```
SELECT A.*
  FROM
    Applicant A
    INNER JOIN Document D
        ON D.applicant_id = A.applicant_id
    INNER JOIN Term T
        ON T.document_id = D.document_id
  WHERE
    T.term = 'CPA'
```

Note that because of the join syntax here, and the fact that we have normalized the relationships, this query will actually return one row per applicant *per document* -- so, if one applicant uploaded two resumes, both containing the word CPA, then we would get two results. It is unlikely that this is the result that our HR manager is looking for, so to fix the query to "hide" this normalization we have done, we might use the DISTINCT operator to transform the "bag" of Applicants back to a proper set of applicants, as in “SELECT DISTINCT A.* ...”. Or, to express the query more directly in terms of our intention, we might use a semijoin with the "EXISTS" operator:

```
SELECT *
  FROM
    Applicant A
  WHERE
    EXISTS (
        SELECT *
          FROM
            Document D
            INNER JOIN Term T
                ON T.document_id = D.document_id
          WHERE
            D.applicant_id = A.applicant_id
            AND T.term = 'CPA'
    )
```

In either case, this is not a trivial query to write correctly, because of the complexity with which we have described the relationships between applicants, documents, and the terms in those documents. Additionally, if the number of applicants is high, we might get into situations where the performance of this design is very challenging; indexes to support millions of terms and thousands of applicants are well within the purview of today's commercial relational database systems, but millions of users with tens of billions of terms might cause a bit more headache.

How might this scenario be more naturally modeled in the non-relational world? We turn to the Cassandra project for this example, using column families and super-columns (described in more detail below). A possible design for the applicant table might be:

Applicant Row	Column Families	
	Supercolumn Answers:	Supercolumn Terms:
<question_id>	**Answer:<question_id> =answer**	**Term:<document_id> =term**

This design groups all of the information about the applicant—their answers to questions, as well as the keyword terms in their attached documents—into a single entity of the data store; however, each column family may be distributed separately, and the supercolumns within the family can contain any number of values, each of which can be versioned any number of times. This simple multidimensional approach provides locality of the data and high performance with regard to the physical storage properties, but perhaps more importantly, it simplified the nature of the data definition by describing it physically in much the same way you might think about it logically.

How would you query this data? Certainly not with a standard SQL query, which has no way of interacting with the nested properties of the data. Instead, custom procedural code would have to be written to fetch the desired records, iterate over them, and produce the results.

Is this more or less difficult than the SQL queries shown above? In the simplest real-world case of slip-shod requirements and quick turnaround time, the answer is probably that the SQL queries are simpler to write. However, the other properties of the data access may shift the balance of this equation; when the task is not to produce a quick report, but instead to manage this information for millions of users, in order to produce intermediate structures that can answer search queries in fractions of a millisecond, the prospect of writing your own access code in this manner (via, for example, a map/reduce operation) becomes much more attractive.

SECTION 3: BENEFITS

It should be clear at this point that there are trade-offs in the expressive power of relational versus non-relational data stores, depending intimately on the problem domain being modeled. With that in mind, the next two sections present a more formal set of dimensions which might be reasonably considered "benefits" of non-relational database modeling, and subsequently, "detriments" of non-relational database modeling. These include inquests into the expressive power of the data modeling abstractions provided by the systems, as well as more particular concerns about the integrity requirements and access patterns of applications.

There are a long list of potential advantages to using non-relational databases. Of course, not all non-relational databases are the same; but the following list covers areas common to many of them.

- Semi-Structured Data
- Alternative Model Paradigms
- Multi-valued properties
- Generalized Analytics
- Version History
- Predictable Scalability
- Schema Evolution

We will explore each of these areas in turn.

SEMI-STRUCTURED DATA

We saw above the value of the "Expando" concept from the Google App Engine data store Python API - a structure where each entity can have any number of properties defined at run-time. This approach is clearly helpful in domains where the problem is itself amenable to expansion or change over time (as were the Questions related to our Positions). We can begin simply, and alter the details of our problem as we go with minimal administrative burden. This approach has much in common with the imputed typing systems of scripting languages like Python, which, while often less efficient than strongly typed languages like C and Java, usually more than make up for this deficiency by giving programmers improved usability; they can get started quickly and add structure and overhead only as needed.

But there is another, more important aspect to this tendency towards storing non-structured, or semi-structured, data: the idea that your understanding of a problem, and its data, might *legitimately* emerge over time, and be entirely data-driven after the fact. As one observer put it:

> RDBMSs are designed to model very highly and statically structured data which has been modeled with mathematical precision - data and designs that do not meet these criteria, such as data designed for direct human consumption, lose the advantages of the relational model, and result in poorer maintainability than with less stringent models. [Barreto, 2009]

This kind of emergent behavior is atypical when dealing with the programming problems of the past 40 years, such as accounting systems, desktop word processing software, etc. However, many of today's interesting problems involve unpredictable behavior and inputs from extremely large populations - consider web search, social

network graphs, large scale purchasing habits, etc. In these "messy" arenas, the impulse to exactly model and define all the possible structures in the data in advance is exactly the wrong approach. Relational data design tends to turn programmers into "structure first" proponents, but in many cases, the rest of the world (including the users we are writing programs for) are thinking "data first".

There is a negative side to this tendency as well, of course; we will return to that in the next section.

ALTERNATIVE MODEL PARADIGMS

Modeling data in terms of relations, tuples and attributes--or equivalently, tables, rows and columns--is but one conceptual approach. There are entirely different ways of considering, planning, and designing a data model. These include hierarchical trees, arbitrary graphs, structured objects, cube or star schema analytical approaches, tuple spaces, and even undifferentiated (emergent) storage. By moving into the realm of semi-structured non-relational data, we gain the possibility of accessing our data along these lines instead of simply in relational database terms.

For example, there is an entire class of non-relational database systems that we have not talked about in this paper, but that deserves mention: graph-oriented databases, such as Neo4j. This paradigm attempts to map persistent storage capabilities directly onto the graph model of computation - sets of nodes connected by sets of edges. The database engine then innately provides many algorithmic services that one would expect on graph representations: establishing spanning trees, finding shortest path, depth and breadth-first search, etc.

You could certainly model a graph in any relational database; in fact, you need only two relations:

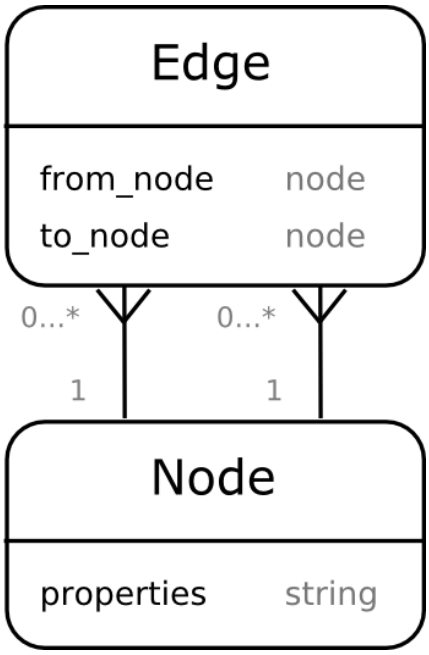

Figure 13: Relational model of a graph

The issue with taking this approach, however, and the advantage of using a full-fledged graph-oriented database, is that the basic operations one might want to use on graph data are entirely different from those available in a SQL paradigm. It would take a recursively defined SQL query to find, for example, a path between two arbitrary nodes. A native graph database, on the other hand, will have primitives for such things built into its query language, coupled with efficient implementations of these operations in terms of indices, disk i/o, etc.

Do other non-relational databases (not specifically geared towards graph problems) achieve this same benefit? To some degree, they do, insofar as they imply a step away from the limitations of SQL. When interactions with a data store imply a map/reduce query architecture, the process of constructing a graph in memory and

working with it becomes just another possibility in the design space. (That said, for large graphs, there may be cases where a map/reduce paradigm is not the most efficient way to interact with the graph.)

Object databases are another paradigm that have, at various times, appeared poised to challenge the supremacy of the relational database. An example of a current contender in this space is Persevere (http://www.persvr.org/), which is an object store for JSON (JavaScript Object Notation) data. Advantages gained in this space include a consistent execution model between the storage engine and the client platform (JavaScript, in this case), and the ability to natively store objects without any translation layer.

Here again, the general principle is that by moving away from the strictly modeled structure of SQL, we untie the hands of developers to model data in terms they may be more familiar with, or that may be more conducive to solving the problem at hand. This is very attractive to many developers:

> The main reason why relational databases are so effective and why programmers hate them so much is that they are data-centric. Programmers tend to see data as secondary or peripheral to code. This programmer bias is the main fuel in the quest for something "better" than an RDBMS, resulting in reinventing wheels that were partially or completely rejected in the 1970s (such as the hierarchical model). [Bain, 2009]

MULTI-VALUED PROPERTIES

Even with the bounds of the more traditional relational approach, there are ways in which the semi-structured approach of non-relational databases can give us a helping hand in conceptual data design. One of these is by way of multi-value properties—that is, attributes that can simultaneously take on more than one value.

A credo of relational database design is that for any given tuple in a relation, there is only one value for any given attribute; storing multiple values in the same attribute for the same tuple is considered very bad practice, and is not supported by standard SQL. Generally, in cases where one might be tempted to attempt to store multiple values in the same attribute, that is a sign that the design needs further normalization.

As an example, consider a "user" relation, with an attribute "email". Since people typically have more than one email address, a simple (but wrong, at least for relational database design) decision might be to store the email addresses as a comma-delimited list within the "email" attribute:

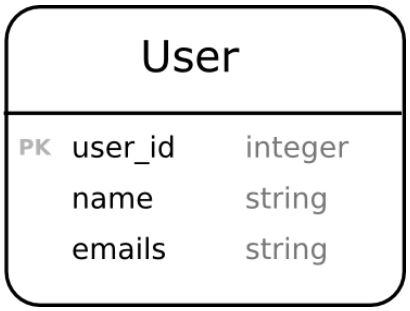

Figure 14: User / email denormalized model

Example data in the table might include:

```
user_id   name            emails
--------- --------------- ----------------------------------------------------
123       Homer Simpson   homer@simspon.com, homer.simpson@springfieldpower.org
```

The problems with this are myriad - for example, simple membership tests like

```
SELECT * FROM User WHERE emails = 'homer@simpson.com'
```

will fail if there are more than one email address in the list, because that is no longer the value of the attribute; a more general test using wildcards such as:

```
SELECT * FROM User WHERE emails LIKE '%homer@simpson.com%'
```

will succeed, but raises serious performance issues in that it defeats the use of indexes and causes the database engine to do (at best) linear-time text pattern searches against every value in the table. Worse, it may actually impact correctness if entries in the list can be proper substrings of each other (as in the list "car, cart, art").

The proper way to design for this situation, in a relational model, is to normalize the email addresses into their own table, with a foreign key relationship to the user table, like so:

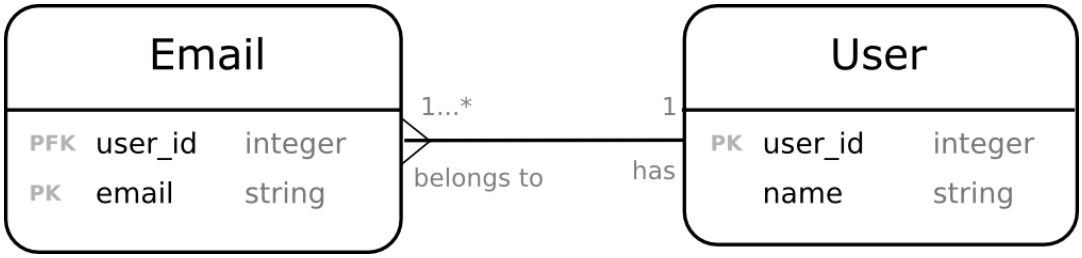

Figure 15: Normalized model of user with emails

This is the standard Many-to-one design pattern we say early in the introductory examples (in that case, between Applicants and Positions). The same data would thus be rendered in this model as follows:

```
user_id     name
----------- ---------------
123         Homer Simpson

user_id     email
----------- ------------------------------------------------------
123         homer@simspon.com
123         homer.simpson@springfieldnuclear.org
```

This is a design strategy that can is frequently applied to many situations in standard relational database design, even recursively: if you sense a one-to-many relationship in an entity, break it out into two relations with a foreign key.

The trouble with this pattern, however, is that it still does not elegantly serve all the possible use cases of such data, especially in situations with a low cardinality - either it is overkill, or it is a clumsy way to store data in certain situations. In the above example, there are a very small set of use cases that we might typically do with email addresses, including:

- return the user with their one "primary" email address, for normal operations involving sending an email to the user
- return the user with a list of all their email addresses (e.g. for showing on some kind of "profile" screen)
- find which user (if any) has a given email address

The first situation requires an additional attribute along the lines of "is_primary" on the email table, not to mention logic to ensure that only one email tuple per user is marked as primary (which cannot be done natively in a relational database, because a UNIQUE constraint on the user_id and the is_primary field would only allow one primary and one non-primary email address per user_id). Alternately, a "primary_email" field can be kept on the User table, acting as a cache of which email address is the primary one; this too requires coordination by code to ensure that this field actually exists in the User_Email table, etc.

To use standard SQL to return a single tuple containing the user and all of their email addresses, comma delimited like our original ("wrong") design concept, is actually quite difficult under this two-table structure. For example, if our desired output is:

```
user_id  name            email
-------- --------------- ----------------------------------------------------
123      Homer Simpson   homer@simspon.com, homer.simpson@springfieldpower.org
```

Standard SQL has no way of rendering this output, which is surprising considering how common it is. The only mechanisms would be constructing intermediate temporary tables of the information, looping through records of the join relation and outputting one tuple per user_id with the concatenation of email addresses as an attribute.

Under key/value stores, we have a different paradigm entirely for this problem, and one which much more closely matches the real-world uses of such data. We can simply model the email attribute as a substructure: a list of emails within the attribute. The logical model is as simple as:

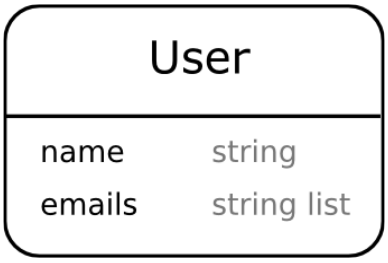

Figure 16: Non-relational model of user and emails

For example, Google App Engine has a "List" type that can store exactly this type of information as an attribute:

```
class User(db.Model):
    name = db.StringProperty()
    emails = db.StringListProperty()
```

(As before, we have removed the "id" attribute, as that is handled by the "key" of the entity instances.)

The query system then has the ability to not only return the contained lists as structured data, but also to do membership queries, such as:

```
results = db.GqlQuery("SELECT * FROM User WHERE email = 'homer@simpson.com'")
```

This will return user 123, because it returns any instances where any of the values in the list match the query.

Since order is preserved, the semantics of "primary" versus "additional" can be encoded into the order of items, so no additional attribute is needed for this purpose; we can always get the primary email by saying something like "results.emails[0]".

In effect, we have expressed our actual data requirements in a much more succinct and powerful way using this notation, without any noticeable loss in precision, abstraction, or expressive power.

GENERALIZED ANALYTICS

On the subject of expressive power, consider again the "GROUP BY" example above. Our use of SQL in this case was standard and straightforward; “GROUP BY” is a SQL primitive, and allows one level of aggregation, by one or more attributes. If the analytics you are performing falls into this category, it is difficult to argue that there is a more succinct way to express it than this.

However, as explained above, if the nature of the analysis falls outside of SQL’s standard set of operations, it can be extremely difficult to produce results with the operational silo of SQL queries. Worse, this has a pernicious effect on the mindset of data developers, sometimes called “SQL Myopia”: if you can’t do it in SQL, you can’t do it[11].

[11] Note that this is not a fault of the relational model itself—only of SQL, which is ultimately just one possible declarative grammar for interacting with relational structures.

This is unfortunate, because there are many interesting and useful modes of interacting with data sets that are outside of this paradigm – consider matrix transformations, data mining, clustering, Bayesian filtering, probability analysis, etc.

Additionally, besides simply lacking Turing-completeness[12], SQL has a long list of faults that non-SQL developers regularly present. These include a verbose, non-customizable syntax; inability to reduce nested constructions to recursive calls, or generally work with graphs, trees, or nested structures; inconsistency in specific implementation between vendors, despite standardization; and so forth. It is no wonder that the moniker for the current non-relational database movement is converging on the tag "NOSQL": it is a limited, inelegant language.

Non-relational databases skirt the entire issue by requiring most interactions with the data store to be written in other conventional languages. This opens up the possibilities of what can be done with data (though it also has negative implications in terms of ease of use, as we will explore below).

VERSION HISTORY

Part of the design of many (but not all) non-relational databases is the explicit inclusion of version history in the storage unit of data. For example, when you store the value 123 in an attribute, and later change it to the value 234, your data store actually now contains both values, each with a timestamp or vector clock version stamp. This approach has many benefits from an efficiency point of view: primary interaction with the database disks is always in write-forward mode, and multi-version concurrency control can be easily modeled with this structure.

[12] For the record, this lack of Turing-completeness is by design, so that all queries would be able to run in bounded time; never mind that every major commercial vendor has extended SQL with operations that do make it Turing complete, albeit still awkward.

From a modeling point of view, however, there are other distinct advantages to this format. One of them is the ability to intentionally keep, and interact with, older versions of data in a structured way. An example of this, which almost certainly uses the versioned characteristics of Google's BigTable infrastructure, is Google Docs: any document can be instantly viewed in, or reverted to, its state at any point in its history – a granular, infinite "undo".

Implementing this kind of revision ability in typical relational database applications is prohibitive both from a programming complexity standpoint (this ability must be consciously designed in to each entity that might need it) as well as from a performance standpoint[13].

As an example of this difficulty, consider the options we would have if we wanted to be able to version the data in our example Applicant table above—for example, if government non-discrimination regulations required our HR department to show a full audit trail on any changes made to applicant data. The basic (original) logical design of the Applicant relation:

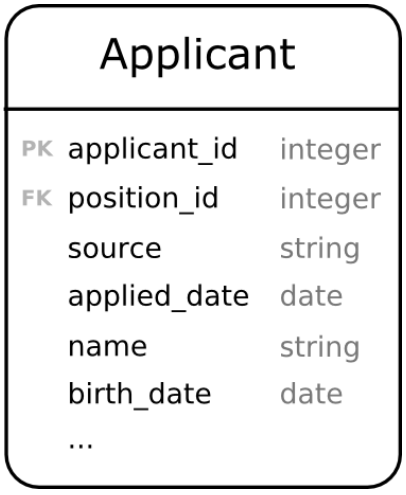

Figure 17: Applicant entity

[13] Consider how many traditional relational database implemented products you know of that offer any kind of Undo functionality.

We have two main options when keeping a history for information in this table. On the one hand, we can keep a full additional copy of every row whenever it changes. This can be done in place, by adding an additional component to the primary key which is a timestamp or version number:

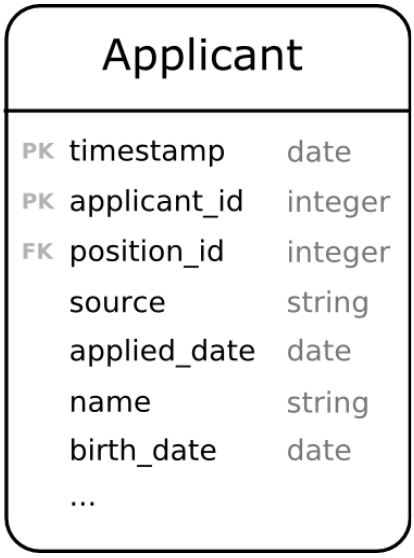

Figure 18: Applicant history table with timestamp

This is problematic in that all application code that interacts with this entity needs to know about the versioning scheme; it also complicates the indexing of the entities, because relational database storage with a composite primary key including a date is significantly less optimized than for a single integer key.

Alternately, the entire-row history method can be done in a secondary table which only keeps historical records, much like a log:

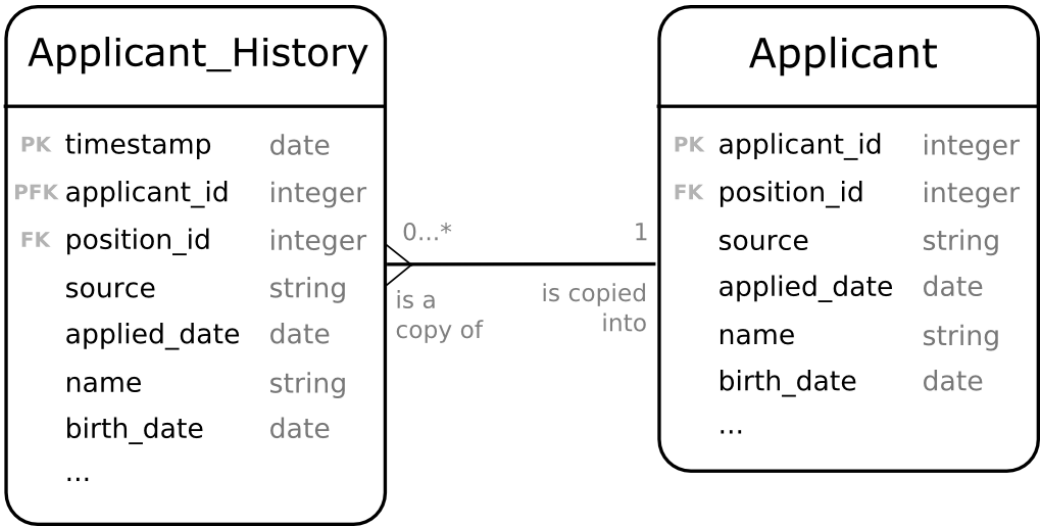

Figure 19: Historical versions implemented as an additional table

This is less obtrusive on the application (which need not even be aware of its existence, especially if it is produce via a database level procedure or trigger), and has the benefit that it can be populated asynchronously.

However, both of these cases require O(s*n) storage, where s is the row size and n is the number of updates. For large row sizes, this approach can be prohibitive.

The other mechanism for doing this is to keep what amounts to an Entity / Attribute / Value table for the historical changes: a table where only the changed value is kept. This is easier to do in situations where the table design itself is already in the EAV paradigm, but can still be done dynamically (if not efficiently) by using the string name of the updated attribute:

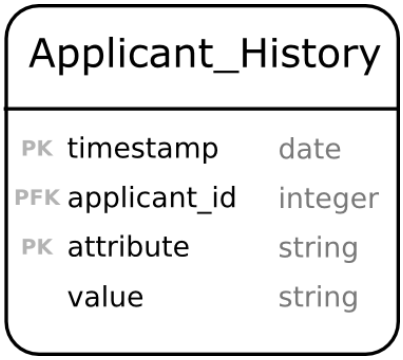

Figure 20: Historical version using an entity/attribute/value model.

For sparsely updated tables, this approach does save space over the entire-row versions, but it suffers from the drawback that any use of this data via interactive SQL queries is nearly impossible, owing to the same SQL complexities we saw above when examining use of the EAV model—compounded now by the addition of a time component.

Overall, the non-relational database stores that support column-based version history have a huge advantage in any situations where the application might need this level of historical data snapshots.

PREDICTABLE SCALABILITY

While the focus of this report is not on the implementation-specific aspects of scalability, it is important to note that one of the most important benefits of this class of data store - and in fact, the justification for their existence in the first place - is their ability to scale to larger, more parallel installations than relational databases can.

This definitively impacts the modeling concepts supported by the systems, because it elevates scalability concerns to a first class modeling directive - part of the

logical and conceptual modeling process itself. Rather than designing an elegant relational model and only later considering how it might reasonably be "sharded" or replicated in such a way as to provide high availability in various failure scenarios (typically accompanied by great cost, in commercial relational database products), instead the bedrock of the logical design asks: how can we conceive of this data in such a way that it is scalable by its definition?

As an example, consider the mechanism for establishing the locality of transactions in BigTable and its ilk (including the Google App Engine data store). Obviously, when involving multiple entities in a transaction on a distributed data store, it is desirable to restrict the number of nodes who actually must participate in the transaction. (While protocols do of course exist for distributed transactions, the performance of these protocols suffer immensely as the size of machine cluster increases, because the risk of a node failure (and thus a timeout on the distributed transaction) increases.) It is therefore most beneficial to couple *related* entities tightly, and unrelated entities loosely, so that the most common entities to participate in a transaction would be those that are already tightly coupled. In a relational database, the relationships you might to indicate related entities are foreign key relationships, but that relationship itself carries no additional information that might indicate "these two things are likely to participate in transactions together". By contract, in BigTable, this is enabled by allowing entities to indicate an "ancestor" relation chain, of any depth. That is, entity A can declare entity B its "parent", and henceforth, the data store organizes the physical representation of these entities on one (or a small number of) physical machines, so that they can easily participate in shared transactions. This is a natural design inclination, but one that is not easily expressed in the world of relational databases (you could certainly provide self-relationships on entities but since SQL does not readily express recursive relationships,

that is only beneficial in cases where the self-relationship is a key part of the data design itself, with business import.)

Many commercial relational database vendors make the claim that their solutions are highly scalable. This is true, but there are two caveats. First, of course, is cost: sharded, replicated instances of Oracle or DB2 are not a cheap commodity, and the cost continues as the load increases. Second, however, and less obvious, is the predictability factor. This is highly touted by systems such as Project Voldemort, which point out that with a simple data model as is exposed in many non-relational databases, not only can you scale more easily, but you can scale more *predictably*: the requirements to support additional operations, in terms of CPU and memory is known fairly exactly, so load planning can be an exact science. Compare this with SQL / relational database scaling, which is highly unpredictable due to the complex nature of the RDBMS engine. To wit:

> Voldemort queries have known performance, so it is very easy to predict the load a new feature will generate by just counting the number of requests. This is always a challenge with SQL: poorly designed SQL queries may produce thousands of times more load. Compounding this problem, distinguishing the bad queries from the good requires knowing both the index structure and the data on which it will run—neither of which is present in your code—so it easy for an efficiency to slip past even a diligent review if you don't perform real tests on real data for each modification to see what query plan will be generated. [Kreps, 2009]

There are, naturally, other criteria that are involved in the quest for performance and scalability, including topics like low level data storage (b-tree-like storage formats, disk access patterns, solid state storage, etc); issues with the raw networking of systems and their communications overhead; data reliability, both considered for single-node and multi-node systems, etc. Some issues in this arena will be touched on below in the Survey section with regard to individual implementations.

SCHEMA EVOLUTION

In addition to the static existence of a database schema, it is also important to consider what happens over time as an application's needs or requirements change Non-relational databases have a distinct advantage in this realm, because they offer more options for how the version update should proceed [Strauss, 2009].

To be sure, relational databases have mechanisms for handling ongoing updates to data schema; indeed, one of the strengths of the relational model is that the schema *is* data – databases keep system tables which define schema metadata, which are handled by the exact same database primitives as user-space tables. This generality has advantages in terms of manageability, but it also provides a clean abstraction that vendors can use to provide valuable schema update facilities. Indeed, commercial RDMBS products have applied a great deal of engineering resources to the problem, and have developed sophisticated mechanisms that allow production databases to ALTER their schema without downtime in most scenarios[14]. However, there are two issues with the relational database approach to this.

First, relational database schemas exist in only one state at any given time. This means that if the specific form of an attribute changes, it must change immediately for all records, even in cases where the new form of the attribute would rightfully require processing that the database cannot do (for example, application-specific business logic). It also implies that if there is a high-volume update, such as one that might need to write many gigabytes of changed data back to disk, the RDBMS is obligated to do this operation atomically and in real-time (because DDL updates are transactional); regardless

[14] Non-commercial databases such as MySQL also have mechanisms such as this, but as of this writing, in general their methods are much less sophisticated, often requiring downtime to do even simple operations such as rebuild indices, etc. See [Taylor, 2009] for examples.

of how efficiently implemented it is, this type of operation cannot be made seamless in a highly transactional production environment.

Second, the release of relational database schema changes typically requires precise coordination with application-layer code; that is, the code version must exactly match the data version. In any highly available application, there is a high likelihood that this implies downtime[15], or at least advanced operational coordination that takes a great deal of precision and energy.

Non-relational databases, by comparison, can use a very different approach for schema versioning. Because the schema (in many cases) is not enforced at the data engine level, it is up to the application to enforce (and migrate) the schema. Therefore, a schema change can be gradually introduced by code that understands how to interact with both the N-1 version and the N version, and leaves each entity updated as it is touched. "Gardener" processes can then periodically sweep through the data store, updating nodes an a lower-priority process.

Naturally, this approach produces more complex code in the short term, especially if the schema of the data is relied upon by analytical (map/reduce) jobs. But in many cases, the knowledge that no downtime will be required during a schema evolution is worth the additional complexity. In fact, this approach might be seen to encourage a more agile development methodology, because each change to the internal schema of the application's data is bundled with the update to the codebase, and can be collectively versioned and managed accordingly.

---

[15] The exception to this is that, thanks to the relational model's implicit lack of attribute order, there are situations in which new attributes can be added and it is guaranteed that no application code would even know of the existence of the new attributes, let alone be affected by them. This is a case where the relational model has the upper hand; however, because it is not a comprehensive solution for every situation, the end result is that, for safety, most relational database schema updates are treated as downtime events.

SECTION 4: DETRIMENTS

> There are a few new ideas in storage systems these days, but many of them are bad ideas, and many things that were good in relational databases have been lost.
>
> - Jay Kreps, Project Voldemort Author

Having explored the conceptual gains we get from using key/value models of data design, we now turn out attention to the darker side: the benefits of relational databases that we lose when moving to their non-relational cousins.

We noted above that since any arbitrary computation can be layered on top of non-relational data stores, we can potentially emulate any of the behaviors of a relational database in application code. This is certainly true, but the statement belies a misunderstanding about the true complexity, and value, of the services built into today's relational databases. This section explores those areas that are a) not currently well supported at the data level, and b) would be non-trivial to replicate in application code. These include:

- Ease of expression - writing queries is fast and easy, assuming those requirements are within the purview of what SQL can do natively.
- Concurrency and Transactions - ACID properties
- Eventual Consistency
- Normalized Updates and relational integrity
- Standardization
- Access Control

EASE OF EXPRESSION

As we saw above, standard SQL is not a Turing-complete language; there are many concepts that cannot be expressed eloquently, or at all. It is also a somewhat antiquated language, lacking the modern niceties of object-orientation, robust debugger support, etc.

Separate from that list of complaints, however, it is important to note that for the things it can do well, SQL is an extremely concise declarative language; it builds a consistent, useful abstraction framework on top of data storage in the relational model, and allows implementations to optimize access to the data within the bounds given by that abstraction. This has significant benefits in terms of the ease with which developers can do common (and many uncommon) tasks.

For one thing, it is effectively impossible to have low-level bugs in SQL code. That is not to say that there are not high level bugs—an incorrect join, a wrong assumption about a data model's properties, etc. But it is impossible to have an error in the JOIN operator or the sorting algorithm, because these are system-standard components that are accessed only declaratively. Conversely, when it is up to the programmer to correctly (and efficiently!) implement all of these operations each time they are needed, that opens the door to a huge class of problems that simply do not exist when working with relational databases. It is not necessary to test whether the math performed by the aggregation engine using a GROUP BY statement is correct; it is.

Non-relational stores generally allow queries against only the primary key of the store, possibly with one additional layer of filtering via index to limit results to only those that match a simple set of filters (i.e., WHERE clauses). This limitation is acceptable in many cases, but it is important to note that a drastic departure from SQL it really is; SQL

allows an arbitrary complexity of query syntax, and relational databases management systems typically have an incredibly complex layer for processing and planning the execution of these (potentially complex) queries. Nested queries, complex table joins, aggregation and pivoting, projections—all can be described in SQL, and a good query processing system will quickly craft extremely efficient mechanisms for answering these queries. For SQL-friendly data access patterns, a good SQL programmer can create data access and manipulation code far faster than in any other language, because the set-based operations are logical, clean, and declarative. That doesn't guarantee that these patterns will be the most high-performing, but it's likely they will be at least competitive, because they implicitly take advantage of all the engineering that has been done within the database engine, which often includes extreme but subtle optimizations that would be very difficult to replicate quickly.

Of course, as we showed in the earlier examples involving analytic workloads, there is also the opposite effect, summed up by the phrase "If the only tool you have is a hammer, then every problem looks like a nail." If your analyses are limited to what can comfortably and easily be expressed in SQL, there is a wide range of possible abilities that you are overlooking.

UNDERSTANDING YOUR DATA

Above, we touted semi-structured data as a benefit of non-relational databases: get started quickly, don't spend time creating elaborate relational schemas. This approach appears to be heavily favored by some of the vendors who offer non-relational solutions. In fact, much of the language is distinctly hyperbolic, offering to "eliminate the administrative burden of data modeling" [Amazon.com, 2009]. While few will argue that modeling complex data is always fun, reducing it to an "administrative burden" overlooks

the essential qualities of data modeling as a doorway to understanding the related nature of data in any domain.

Any serious system design effort that deals with persistent data must take careful consideration of what the *nature* of that data is. What are the entities? What are the attributes and relationships? Logical data modeling (in UML or otherwise) can often be a very helpful step in understanding the needs of the users and possible overarching system organizational patterns.

At a more tactical level, there are also some advantages to giving constrained schemas to your data. Having no schema also means no protection against mistakes - misspellings, for example:

> In SimpleDB, you are working without the safety net of a predefined schema, and the service will not alert you if you make a mistake. Without a safety net, it could prove to be very painful if you fall. [Murta 2008]

To be fair, few of the solutions in the non-relational space claim that their approach should be jumped into with no forethought; in fact, most of them assume a significantly advanced developer skill set, including the ability to write map/reduce operations, sometimes in new and uncommon functional languages such as Erlang. This is part of the explicit trade-off of these systems: the database engine gives you more control and less of a safety net, in exchange for advanced abilities to scale and perform.

CONCURRENCY AND TRANSACTIONS

Any multi-user data storage engine must deal with issues of concurrency: what happens when two users attempt to change the same value at the same time? The phrase "same time" here may be a bit misleading, in that a single *instant* in time is not implied; any overlapping spans of time have the capability to cause concurrency contention; user

A begins a "read / modification" cycle taking some span of time, and partway through that span of time, user B begins a conflicting "read / modification" cycle. The goal of any such action, from the point of view of the database system, is to make the entire sequence "serializable" - that is, identical to what it would have been had the transactions been placed end-to-end, with no overlapping time span. The more transparently this can be done, the better the throughput of the application will be: time spent waiting for concurrent writes to complete amounts to additional latency in the overall performance of the application.

This may seem to be an esoteric subject, in that locking and concurrency on modern machines might imply extremely fine intervals that would never in practice be violated. But, as Amazon.com's Werner Vogels says:

> " … when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and need to be accounted for up front in the design and architecture of the system." [Vogels, 2008]

Relational databases traditionally use a mechanism known as locking, or "pessimistic" concurrency control; a transaction will identify the resources it intends to change, and protect these resources with a lock (of which there may be various types, depending on the specifics of the operation). Other transactions wishing to update the same resource must wait for the lock to be released. Participants wait their turn for exclusive access to the data, and then commit (assuming they are not involved in a deadlock, where two separate transactions attempt to incrementally incorporate resources already held by the other—a situation which must be separately recognized and resolved by the storage engine itself).

Locking is often the most high-performing approach, because while there is overhead to the locking mechanism itself, it is outweighed by the alternative of transactions failing repeatedly due to high concurrency. Locking does suffer from two problems that are critical from the perspective of non-relational database management systems, however: first, they impose overhead, which is itself anathema to the project of these lean databases; the credo of these systems is typically "store my data with the minimum amount of overhead, and I'll worry about everything else". From that perspective, even the modest overhead of a locking mechanism might be seen as too onerous. More important, though, is that locking is much more difficult to do correctly if the participants in the transaction are distributed - protocols do exist that can provably establish and release locks correctly in a distributed system [Bernstein, 1981], but they are a) slow, and b) even slower in the presence of possible node failures. For this reason, locking is not used by any of the distributed non-relational database systems we survey in this paper, and many architects even shy away from proven distributed transaction techniques such as Paxos and 2PC because of their fragility and poor performance characteristics [Helland, 2007].

As an alternative, another form of concurrency control is typically used in non-relational databases: Optimistic Concurrency, also known as MVCC (Multi-Version Concurrency Control). This mechanism relies on timestamps (presupposing a shared clock) or Vector Clocks, as described in [Lamport, 1978], to determine the modification dates of transactions. In a nutshell, when transaction A begins, it reads the timestamps of the entity or entities it wishes to modify. It then does its computations, and prepares its write. Just before writing, it checks the timestamp of the values again and looks to see if a conflicting transaction (transaction B) has updated the values. If so, the write would be in conflict, and its changes are rolled back and forced to start again from scratch.

Optimistic Concurrency has several properties that make it an ideal choice for large scale distributed database implementations. In opposition to locking mechanisms, reads are never blocked, which can be important if the access pattern of the application calls for large amounts of reads (as many queries in the map/reduce paradigm do). MVCC is very good at achieving true "snapshot" isolation, because a query can carry with it a timestamp that is used to filter any entity the query touches; this is true not only in short terms "near" queries, but also equally effective in reconstructing historical snapshots. Other methods of concurrency control, such as locking, typically impose very high performance costs for doing this.

Using Optimistic Concurrency, however, may introduce additional layers of complexity to the program code, which would be silently handled in relational databases. When one thread is attempting to modify data in a transaction, any concurrent attempts to update the same data will either be forced to retry (which might be built in to the database engine, or else must be implemented by the application developers) or else fail completely; the application can attempt a write again, perhaps up to a preset number of retries before reporting failure, or alternately using some kind of back-off scheme.

The result of this restriction is that in most non-relational database systems, explicit (multi-step) transaction either do not exist at all, or have various limits placed on what they can do. As an example, Google App Engine Data Store can do transactions, but not arbitrary transactions: entities must be declared to be part of the same "entity group" at design time, which is a signal to the data store engine to store the entities in a way that supports transactions, which presumably says something about the particular disk storage and locality of the data within the storage engine clusters.

This is not entirely a bad property, however; it could be argued that a relational database's ability to silently handle such situations causes applications to be designed

with "bottlenecks" that do not become obvious until such time as transactional throughput increases to high levels, at which point there is no simple way to re-architect the solution to avoid these bottlenecks. If instead, the platform itself required data designers to carefully consider which elements might be the source of high contention, and explicitly design around this fact, then the addition of greater load would be less likely to throttle the performance of the application. In fact, this promise - that once you design an application, you will never need to worry about scaling it - is the underlying premise of the marketability of cloud computing solutions such as Amazon SimpleDB, Google App Engine, etc. They are able to make this promise, in part, because of simple design restrictions such as this one.

In the simplest implementation of optimistic concurrency, there is one caveat. If the model is to a) get a timestamp, b) prepare the updates, c) check that the timestamp is unmodified, and d) write the updates - if steps c and d are not done automatically, there is a chance - albeit slight - that consistency is actually broken, because another transaction could theoretically write to the database in between steps c and d. Thus, unless you are able to enforce the atomicity of those two operations - via a lock, a token, etc. - then there is always the possibility of inconsistent data. For some applications, this is not problematic; however, for applications where the success of the software relies on ultimate, inviolate consistency of the database, this is not an option.

CONSISTENCY

Consistency is the notion (which is often taken for granted in traditional relational database systems) that logically, when a client of a data storage system makes a write to that system, any subsequent read (by that client or others) will get the latest version of that data that was written. At a larger scale than individual data items, this property states

that it should always be safe for clients to assemble any discrete pieces of data they get atomically and have those data items agree in the specific consistent picture of the overall system. Consistency is obviously closely intertwined with the concept of transactionality: concurrent systems require transactional guarantees (at least) in order to maintain consistency.

The trouble with consistency begins when we enter the realm of distributed systems. In [Gilbert, 2002], Brewer's "CAP Theorem" is explored, namely: a distributed system cannot simultaneously support all three dimensions of: *consistency*; *availability* (i.e. for any given response, there is a bounded, and hopefully low, latency for the request to be answered); and *partition tolerance* (the notion that is some portion of the computing resources of the cluster are unavailable, the operation can still complete). This theorem has been proven in the context of distributed system modeling.

Distributed systems (of the type explored in this report, at least) assume partition tolerance; therefore, they must make a choice between consistency and availability. However, few (if any) systems would intentionally design in the possibility of permanent inconsistency (otherwise known as corruption).

Instead, some of the models of non-relational databases use a technique known as "Eventual Consistency" [Vogels, 2008]. The concept does not arise frequently on a single disk system, where typically either your data is consistent, or it is not. Instead, the concept usually applies to cases where a distributed representation of the data is kept - for example, across multiple servers in a cluster. The transaction protocol does not guarantee that reads and writes of all conceivable entities in the database will always be instantaneously consistent. Instead, a weakened guarantee is made: in the case of any sort of failure or latency issues, it is possible that entities may appear temporarily inconsistent, but that they will eventually be made consistent.

While there are certainly areas where eventual consistency can work, there are also cases where it could cause significant problems. Outside of even the financial industries, where the potential problems are obvious, consider any situation where user input is cumulative: that is, user C's update depends on user B's update, which in turn depends on user A's update. If B is temporarily working from an outdated version of A's information, and makes a change which C then acts on, there are any number of scenarios where the important consistency properties of the entire system could be compromised. As such, it is important to carefully consider any part of a model that may run into this type of issue with consistency guarantees.

On the other hand, it is often pointed out that eventual consistency is not a foreign pattern to most people; for example, purchases on a credit card are not typically instantly reflected in the balance, but often take minutes, hours, or days to appear. We will return to an analysis of the factors involved in considering eventual consistency in the Design Strategies section below.

RELATIONAL INTEGRITY

Another issue where we lose confidence when moving to a non-relational data store is in relational integrity; specifically, the ability to enforce, at the database level, that references between entity instances actually refer to real instances of the referenced entity. To return to our running example, in a relational database, if we define a foreign key between the Applicant and Position, we can be sure that the reference is to a real Position that exists; the RDBMS will prevent us from deleting a referenced Position without first deleting (or reassigning) all of the Applicants that point to it (or alternately, if specified, to cascade the delete to related entities). Any attempt to do otherwise will result in an error, and potentially a rolled back transaction.

Can a non-relational database guarantee the same level of protection against integrity problems? Generally speaking, no:

> For constraints to be applied, the tables must reside on a single database server, precluding horizontal scaling as transaction rates grow. […] Schemas that can scale to very high transaction volumes will place functionally distinct data on different database servers. This requires moving data constraints out of the database and into the application. [Pritchett, 2008]

First of all, if the consistency models (as mentioned above) are lax, then the answer is most certainly no; operations could be done referring to entities which have been deleted in one client's view but not another.

But even assuming a stronger consistency model, non-relational databases have a significant amount of work to do if they want to replicate the same level of integrity guarantee that is provided by a relational database. Relational database architectures provide a layer through which all queries are passed, that enforces relational integrity guarantees; this would be extremely difficult to do in a fully distributed environment, and would seriously hamper the system's throughput.

Overall, the declarative constraint language of relational databases more reliably protects against integrity problems than application-level validation, which is subject to coding problems, consistency errors, etc..

In place of proper relational integrity constraints, most non-relational databases offer unenforced references: an entity whose key is used as a reference property in another entity can still be deleted, and it is always up to the application code to check the existence of a referred-to key before proceeding. This is the strategy used, for example, by the Google App Engine Data Store.

Does this matter? That depends greatly on the rest of the system architecture. In this author's experience, it is rare to see cases where production code is written in such a

way as to depend directly on the referential integrity constraints of a DBMS—that is, to intentionally generate and catch foreign key errors as part of the standard operating process. Instead, foreign key constraints are typically more of a fail-safe—a bedrock condition where you know that no matter how badly a software component errs, certain properties of the data are inviolate. This is useful, but is too often used as a crutch where proper system testing would be an equally effective protection.

There is an implicit relationship between relational integrity, transactions, and normalization. Consider a database design for applicants and positions that is denormalized to include both the Position and Applicant attributes in a single entity (as might commonly be done in a non-relational data store):

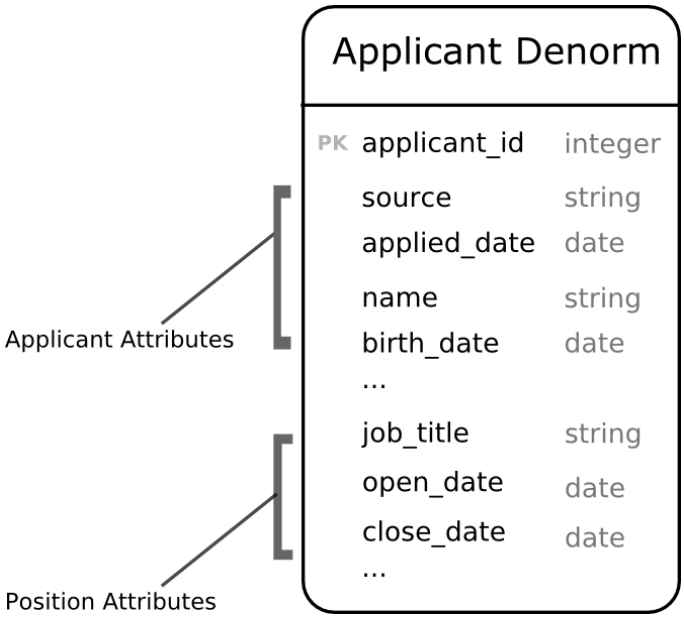

Figure 21: Denormalized Applicant Entity

Obviously, information about the Positions is repeated on each Applicant record in this design. Now imagine that an update must be done to change the title of a position that some large number of Applicants have applied to. Because of the denormalized

design, this requires that all the related Applicant rows be updated. In a relational database with full transaction support, this is no problem at all, even if the data is denormalized—a single UPDATE statement is guaranteed to change the data regardless of its normalization properties, so the two designs (normalized or denormalized) are indeed logically equivalent. Not so for the non-relational store, however: the relaxed transaction guarantees mean that this operation will likely not complete atomically.

STANDARDIZATION

There is rarely an argument for being standardized for its own sake; as they say, "the best thing about standards is that there are so many of them!". However, it is important to consider that in a realm like database storage, adherence to standards (such as SQL, ODBC, etc.) can have unforeseen benefits down the line. Many tools (both commercial and open-source) have extremely high degrees of support for SQL, including automated reporting and visualization, query generation from meta-data, web-based data administration and management, etc. While such layers can of course be written as needed, there is a distinct benefit (at times) to being able to plug into an existing ecosystem of tools and processes (not to mention, skill sets). Stepping outside this comfortably supported zone has its benefits, but also its costs.

Code generally lives longer than expected, and data access code doubly so, because it reflects aspects of the system that are less likely to change as requirements shift slightly. Therefore, the future needs of an application's data are not always clear. For example, in the realm of public companies, the development team may find themselves in an uncomfortable situation with auditors in several years' time, when asked how to query the data related to some control. If the answer is, "just write a distributed map/reduce function in Erlang!", the response from the auditors may not be pleasant.

ACCESS CONTROL

Another category of diminished functionality in the current crop of non-relational databases, compared to most commercial relational databases, is in the area of granular access control. Database systems like Oracle, Microsoft SQL Server, MySQL, etc., all contain a robust security model that allow the creation of user accounts, as well as roles or groups to combine and manage those user accounts. It is then possible to set very detailed, granular permissions regarding which users and / or groups can select, insert, update, and delete data, as well as execute individual functions, procedures, etc. In MySQL, this set of abilities is referred to as "privileges" [MySQL, 2009]. Access control is real-time, meaning that changes to users' and groups' granular access can be changed at any point, and that access is immediately enforced by the database engine itself.

Non-relational stores do not generally provide access control of this granularity[16]. As is the general credo of non-relational systems, granular access control is one more dimension of overhead that large, scalable, distributed database systems can do without.

This discussion also shows off a facet of RDBMS systems that many developers forget about: their capacity to be used by business users, not through pre-written user interfaced, but directly, using the facilities of the system itself -- writing queries, importing data into other tools, etc. There are cases, especially in larger organizations, where the access control primitives of the database management system (restricting certain users to only be able to access certain views, tables, queries, etc.) is a key part of the organization's data dissemination and access control strategy. Building this mechanism up from scratch would be a complex and potentially error-prone effort.

[16] A major exception to this is Google's BigTable, which does enforce access control, but only at the column family level. There are also some research-oriented systems, such as Sun Microsystems' *Celeste*, which do include access controls.

SECTION 5: SURVEY

This section provides a cursory introduction to several existing implementations of non-relational databases.

As mentioned above, the primary focus of these comparisons is expressive power and complexity, not performance per se; rather than exploring the detailed performance characteristics of each system, which would be a massive undertaking in itself, we take it as a given that many of these systems are in use today by companies with extreme data needs, such as Google and Yahoo, precisely because they offer scaling and / or performance benefits above and beyond what any relational database can do.

There are 3 major classes of non-relational databases we will survey:

- DHT key/value stores, including Dynamo, Cassandra, Voldemort, and similar
- Multi-dimensional tabular systems, including Google's BigTable, and open source clones Hypertable and HBase
- Document-oriented databases, including CouchDB and MongoDB

The following sections delve into additional details on each current system, highlighting individual areas where it differs from the pack or offers unusual or elegant ways to handle certain design issues. The first three products surveyed below are “cloud” services, meaning that the entire software and hardware stack for these offerings is hosted with the companies who provide the service, who then charge per usage. The remainder are more traditional server-based products.

GOOGLE APP ENGINE DATASTORE

The Google App Engine is a cloud computing platform - meaning, you can write and upload modules of code to Google's servers, where it will run and serve requests, according to a pre-arranged cost model (free up to a certain point). The Data Store is Google's solution for an integrated database with this cloud computing platform; it is, in essence, a simplified interface to Google's internal storage engine, BigTable[17]. It is referred to in the documentation as "scalable structured storage", and can be accessed using a Python or Java API, through which you can construct queries using an object syntax of a simplified dialect of SQL known as "GQL" [Google, 2009].

The restrictions placed on query plans center on the fact that indexes can be used, but only one pass can ever be made, and no full scans are ever allowed. As such, single ranges can be used if they are ranges on an index, and equality comparisons can be done on any attribute; however, inequality comparisons (!=, <, and >) can only involve one attribute, which must be indexed (because otherwise, the product might be a non-contiguous set of entities).

AMAZON SIMPLEDB / M/DB

SimpleDB is an attribute-oriented key/value database, which is accessed via the "cloud", through the Amazon Web Services platform. As such, it has strict limits in terms of both size and usage; a query can execute for no longer than 5 seconds. Items (records) are limited to 256 attributes (columns), each with a maximum size of 1024 bytes; domains (entities or tables) cannot exceed 10 GB, and entire databases, 1 TB. [Murty, 2008]. From the product documentation:

[17] The reader will already be familiar with the basic concepts of working with the Google App Engine data store, from the examples above.

> A traditional, clustered relational database requires a sizable upfront capital outlay, is complex to design, and often requires a DBA to maintain and administer. Amazon SimpleDB is dramatically simpler, requiring no schema, automatically indexing your data and providing a simple API for storage and access. This approach eliminates the administrative burden of data modeling, index maintenance, and performance tuning. Developers gain access to this functionality within Amazon's proven computing environment, are able to scale instantly, and pay only for what they use. [Amazon.com, 2009]

An interesting aspect of SimpleDB is that it traces some of its lineage to the (mostly academic) concept known as a *tuplespace*, which is a coordination mechanism where collaborators share access to tuples (i.e., records) via a set of atomic read/write primitives, and only those operations may be used to orchestrate shared behavior [Gelernter, 1985].

SimpleDB uses the "eventual consistency" model explained above. Indexes are created on all attributes, which is good for read performance but potentially hazardous to a heavy-write application (though, since the scaling is all done within Amazon's infrastructure, presumably as long as the basic latency is not problematic, this performance aspect is not worrisome). All attributes are stored as strings; this means that if you intend to rely on any ordering other than lexographic - that is, chronological order for dates, or numerical order for numbers - you must encode it correct (for example, by padding your numeric value with a sufficient number of zeros such that all numbers are the same length). The primitive operations are Put, Get, Delete, and Query (which accepts a list of attributes and Boolean operators, in a custom string query format). There is no support for join operations across domains, or (oddly) for sorting results, which must be done in the client process.

Unlike the Google App Engine Data Store, Amazon SimpleDB can be accessed via any application, not just one running in the context of Amazon's entire cloud computing platform.

M/DB is an Open Source pluggable clone of SimpleDB which can be used in substitution with SimpleDB. It is a free alternative, and can be hosted on any local server. This is beneficial in that offers an "escape route" for organizations, should Amazon raise prices or stop offering the SimpleDB service.

MICROSOFT SQL AZURE / DRYAD LINQ

Microsoft actually has two major entries into the cloud-based data storage space, but one of them (**SQL Azure**, formerly SQL Services) is intended to be a full relational database engine running in the cloud, whereas the other (**Windows Azure Storage Service**, formerly Windows Azure Tables) is a simpler, non-relational database offering of the type we are surveying here. This dualism gives some insight into their business strategy in this case:

> Microsoft seems to be alone … in acknowledging that while key/value stores are great for scalability, they come at the great expense of data management, when compared to RDBMS. Microsoft's approach seems to be to strip to the bare bones to get the scaling and distribution mechanisms right, and then over time build up, adding features that help bridge the gap between the key/value store and relational database platform. [Bain, 2009]

The simpler version, Windows Azure Storage Service, offers simple storage of blobs and tables (accessed ISAM-style) in the cloud, as well as cloud-based queues, all available via a RESTful interface. Specific to the Microsoft stack, the main access model of this model is via LINQ (Language INtegrated Query). [Jennings, 2009]

Closer to the computations model of other non-relational engines explored in this report, Microsoft Research has also published research on a model involving accessing Dryad, a distributed execution engine, via LINQ, an inline data specification and access language [Yu, 2008]. This has many potential advantages, including the benefits of declarative SQL programming, a coherent and automated interface into the distribution mechanisms, and good Microsoft tool integration (such as Visual Studio).

BIGTABLE / HYPERTABLE / HBASE

The original published paper on **BigTable**, which is now widely cited in this field of research, is [Chang, 2006]. It laid out the internal organization, and thought processes behind, the large-scale distributed storage that Google implemented to power their extreme storage needs.

In essence, BigTable and its clones are implemented as sparse, multidimensional sorted maps. The three dimensions of any index into this multidimensional array are the row, column, and timestamp; the value is an opaque block of bytes. This model was chosen over a simpler key/value distributed hash table approach because of the advantages it offers for modeling data:

> We believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system. [Chang, 2006]

Rows are the basic unit of atomicity, and updates to a single row are always transactional (which make reasoning about the concurrent properties of the system

manageable for developers). Columns are divided into column families, of which there are a small and static number; column families are the basis for access control, as well as for internal accounting for disk and memory usage. Beyond that, an additional layer called “Locality Groups” was introduced, above the column family layer, to allow developers to indicate which column families were likely to be accessed together, thus giving a hint to the underlying system that these portions of data should be stored together. Bloom filters may be used on top of that to prevent unneeded disk accesses in many cases.

The BigTable paper makes passing mention of what a radical departure their singular data model is from traditional relational database approaches, stopping short of saying that the design is easy to get used to:

> Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice. [Chang, 2006]

Hypertable and **HBase** are two open-source clones of BigTable, both based primarily on the research presented in [Chang, 2006], but also have developed in their own directions since then after having been used in large production environments.

Hypertable is very similar to the design of BigTable. Slight differences are that it uses MVCC (compared to BigTable’s copy-on-write architecture for their memtables) and is architected to run on HDFS (the Hadoop File System) or KFS (compoared to BigTable, which runs on Google’s own GFS).

HBase is another clone, but written in Java instead of C++. This gives it a larger group of available developers to work on it, and a simpler code base, at the expense of

the extreme performance characteristics of both Hypertable and BigTable. They support the same basic data schema, with a couple of interesting additions, like an atomic increment operation, a full web management and monitoring solution, integration with the Hadoop map/reduce framework, rolling upgrade capabilities, and a Non-SQL shell. There are also ongoing development activities towards projects like adding secondary indices, providing object-relational mapping layers, schema management tools, etc.

DYNAMO / DYNOMITE

Next to the BigTable model, Amazon's **Dynamo** [DeCandia, 2007] is the other major research paradigm for non-relational database design. Its model is simpler than that of BigTable: simple key/value pairs, stored in a distributed hash table. There are no joins, no other relational schema – only this basic storage mechanism, with massive scaling abilities, and extraordinarily high availability requirements.

In exchange for this level of scaling and availability, per the CAP theorem [Gilbert, 2002], Dynamo allows applications to relax their consistency guarantees:

> To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use. [DeCandia, 2007]

The major techniques used to make Dynamo work and perform well include:

- *Consistent hashing* – to achieve incremental scalability in the partitioning scheme
- *Vector clocks* – to allow MVCC and read repairs rather than write contention
- *Merkle trees*—a data structure that can diff large amounts of data quickly using a tree of hierarchically hash values
- *Gossip* – A decentralized information sharing approach that allows clusters to be self-maintaining

Dynomite is an open-source implementation of Dynamo, written in Erlang. Erlang is itself an interesting language for such projects, as the entire language is explicitly geared towards supporting concurrency. All thread communication in Erlang is implemented via message passing[18]. It is a functional language, and thus potentially prone to be lower performing than something like C++, but Dynomite appears to already have excellent throughput at this stage in its development.

With both Dynamo and Dynomite, there are a set of tunable parameters, or "knobs", that allow developers to actively make a trade off between availability and consistency. This set of parameters includes:

- **N** – the number of replicas per partition. More replicas means more consistency and durability; fewer means more throughput.
- **R** – the read quorum (i.e. how many identical reads must be done before a value is returned). More reads means more consistency, fewer reads means lower latency.
- **W** – the write quorum (i.e. how many writes must confirm completion before the application will accept the value as having been written). More writers means more consistency, fewer means lower latency.
- **Q –** partitioning factor (a factor of 2). How many nodes will this storage system be distributed over? Fewer means more throughput, more means more availability.

Other implementations in this general family (key/value stores) include Project Voldemort, which is inspired by Dynamo, and Facebook's Cassandra, which is inspired by BigTable. We will look at both of these projects next.

[18] The joke about Erlang is that it "achieves high availability through lowered expectations".

PROJECT VOLDEMORT (LINKEDIN DATA STORE)

Project Voldemort is an application created by the developers at LinkedIn, a popular business-oriented social networking site. As they describe it [Kreps, 2009], it was conceived and started as an add-on to their current IT infrastructure, a research project designed to help them scale certain types of data. Like Amazon's Dynamo, it is a key / value storage system based on consistent distributed hashing, with simple Get / Put / Delete operations. Like Dynamo, it stores multiple versions of each data item, and uses vector clocks for snapshot isolation and for enforcing consistency. If a node has an outdated version of a cell, this can be both discovered and repaired by using the accompanying vector clock information.

An interesting aspect of Project Voldemort is that they chose to implement the on-disk storage mechanism as a pluggable feature of the system—that is, different underlying approaches to storing and retrieving key/value pairs can be used. This allows for a flexible strategy in the face of a) changing application needs and access patterns, and b) changes in the cost/performance characteristics of available secondary storage (for example, opening the door for transparent use of solid state disk drives when they become commercially viable). Then, the layout of records on disk becomes an implementation choice, not an entirely new engineering effort. This is important, because as noted, secondary storage layout schemes are subtle and require a great deal of engineering and testing before they perform optimally.

The consistent hashing algorithms used by Project Voldemort are asymmetrical, meaning that there can be "better" and "worse" nodes in the mix (for example, faster CPUs, more or less memory, etc.), and the hash distribution can account for these differences.

CASSANDRA (FACEBOOK DATA STORE)

Cassandra is the key/value storage engine used in Facebook.com[19], an extremely popular social networking site. A design goal of the project was to enable extremely high write volumes (500M writes per day, for example) without requiring that each write first do an accompanying read. Instead, the idea was to give the system to establish serializability after the fact. [Lakshman, 2009]

Similar to BigTable, Cassandra uses the concept of column families to define data. It also adds the concept of "super columns", which are

The high-availability approach of Cassandra describes itself as "always writeable", meaning that writes never fail. However, subsequent reads can choose to either be "weak" reads (meaning, they may not be consistent) or they can be more poorly performing "quorum" reads (meaning, they go the extra mile to achieve consistency by requiring a quorum of read partitions to agree on the value before reporting it).

Cassandra has an optimized mechanism for handling writes and their subsequent flushing to disk. All writes are first written sequentially in a commit log (similar to the tactic used by relational databases to achieve durability of writes). Then in-memory versions of the updated keys are created, which are periodically saved to disk. Additionally, a bloom filter is used that indicates whether data is (probably) present; this drastically reduces seek operations. There are also periodic disk compaction operations that unify entities spread across nodes.

As a counterpoint to Voldemort's reliance on a pluggable on-disk format layer, Cassandra takes the opposite approach, choosing to maintain strict control over the

[19] Note that there is also an exposed data storage API for Facebook applications known as the "Facebook Data Store". It is not made public to what degree that data store uses Cassandra, but it exhibits similar characteristics.

format of the data on disk. The advantage of this is that when data needs to be copied between nodes (for example, with a new node coming up and entering a cluster), much more efficient means can be used; data can be sent from kernel space directly through a network socket to the network interface of the other machine, which can read it directly into kernel space and write it to disk; no user space operations or other caching layers are ever required, so the operation is extremely efficient. In a situation where nodes enter and leave clusters continually, this level of efficiency does make sense, though it is important to understand the significant engineering challenge this level of optimization presents.

At this time, Cassandra explicitly leaves out support for many database concepts:

* Atomicity guarantees across multiple keys
* Analysis support via Map/Reduce
* Distributed transactions
* Compression support
* Granular security via ACL's

COUCHDB / MONGODB

The remaining two systems we will investigate in depth differ from those we have already seen, in that they are *document-oriented* databases.

CouchDB is the most well-known of such databases. It defines a basic key/value storage mechanism, the target of which is the storage of documents in JSON (JavaScript Object Notation) format. These keys can be stored and read, as in any other system. CouchDB then adds an additional layer by using JavaScript to create persistent views against the stored documents which act like normal database tables and can be queried.

The storage engine for CouchDB does support ACID properties, and its concurrency mechanism is MVCC. It supports RESTful access. At this time, it is not a

truly distributed system, like many others we have seen but they do list scaling via clusters as a future goal; the couchdb-lounge project is a thin layer on top of CouchDB that adds sharding and fault tolerance to CouchDB nodes, and is used in production at meebo.com.

Another interesting goal of CouchDB is to scale *down*; that is, to have an implementation that can run in the context of a mobile phone, a web browsers, etc. This goes hand in hand with the desire to enable the same programming model interface for disconnected operation as for regular operation, which is a particular strength of document-oriented approaches.

A similar project is **MongoDB**. It is a document-oriented database that stores blocks of JSON data, with a stated goal of bridging the gap between key/value stores and relational databases. It does not have ACID or a REST interface, which differs from CouchDB, but has a much more robust query engine. It supports a query language very similar to SQL, instead of map/reduce in JavaScript. It also supports query profiling, replication, indexes, and storage of binary data.

OTHERS

This section mentions a long list of other implementations that are out of the scope of this investigation, but merit mention and some passing remarks.

Concurrent Key/Value Data Stores

- **PNUTS** – Yahoo's Data Store, which has a hybrid map/reduce SQL interface called pig. [Olston, 2008]
- **Tokyo Cabinet / Tyrant**: A transactional key value store, successor to qdbm/gdbm. http://tokyocabinet.sourceforge.net/
- **MemcacheDB** – A persistent key/value store based on Memcached; has transactions for reliability, high availability via replication, and an API w/ many implementations. Used in production by Digg
- **Drizzle** – A scaled down version of the MySQL codebase
- **Schemafree** – A layer that uses a RDBMS to store unstructured data and automatically creates additional tables as indexes into the data blobs http://code.google.com/p/schemafree/
- **Archipelago::Treasure** - A (possibly remote) database that only returns proxies to its contents, and thus runs all methods on its contents itself. Has support for optimistically locked distributed serializable transactions. http://rubyforge.org/projects/archipelago
- **Chord with DHash** - A novel distributed peer to peer hash lookup system, layered with a robust persistence model for key/value data. [Cates, 2003]. http://pdos.csail.mit.edu/chord/
- **Scalaris** - A Dynamo-like scalable, transactional key/value store written in Erlang. http://code.google.com/p/scalaris/

- **Ringo** – An experimental Dynamo-like database, for immutable data. http://github.com/tuulos/ringo/tree/master
- **Redis** - Similar to memcached, but the dataset is non-volatile, and in addition to string values, it can store lists and sets with atomic push / pop operations. http://code.google.com/p/redis/

Embedded Key / Value Stores

- **Berkeley DB, NDBM, GDBM, TDB** - in process key/value databases libraries with DB functionality (locking, crud, etc.)
- **SQLite** – A simpler embedded relational database, with no foreign key support (though it does have ACID properties)
- **hamsterdb.com** – embedded

Object Databases

- **Persevere** - Object DB that provides persistent data storage of dynamic JSON data. http://www.persvr.org/
- **M/DB:X** - http://gradvs1.mgateway.com/main/index.html?path=mdbx - Lightweight JSON / Native XML Cloud database
- **eXist** – XML database

Graph-Oriented Databases

- Neo4J
- AllegroGraph
- Sesame

Research Projects

- **Bayou** - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.5748 - research project published in 1996 with eventual database consistency

- **Celeste** - http://www.opensolaris.org/os/project/celeste/ - "Celeste is a highly-available, ad hoc, distributed, peer-to-peer data store. The system implements semantics for data creation, deletion, arbitrary read and write in a strict-consistency data model."
- **ElasTraS** – An attempt to create a data storage system that is as elastic in its provisioning as other cloud computing resources. [Das, 2009]

Historical Non-relational approaches:

- **GT.M -** a schemaless, hierarchical database with a long and distinguished pedigree in the banking sector. It is a hierarchical associative memory (i.e., multi-dimensional array) that imposes no restrictions on the data types of the indexes and the content - the application logic can impose any schema, dictionary or data organization suited to its problem domain." http://www.fis-gtm.com
- **BTrieve** – Historical (pre-SQL) database management paradigm that used ISAM for raw record management and indexing on disk.
- **LDAP / OpenDS** – Not a general purpose database, but a directory server with database-like properties (can be queried, etc).
- **ESENT (Extensible Storage Engine NT) -** A robust, transactional, semi-structured data store built in to Windows. It is used in Windows software products such as Active Directory and Microsoft Exchange server, and offers ACID properties, snapshot isolation, record-level locking, indexing, complex types such as conditional, tuple, and mult-valued, and is self-adjusting. http://blogs.msdn.com/windowssdk/archive/2008/10/23/esent-extensible-storage-engine-api-in-the-windows-sdk.aspx

SECTION 6: DESIGN STRATEGIES

There are several design points to consider when designing physical models for data to be housed in non-relational databases. This section introduces several overall design strategies, in three parts. First, we present a series of **design questions** that any database designer should ask when beginning a project, which might guide the choice of what paradigm of database modeling should be used. Second, a series of prescriptive **strategies** are given for consideration of data designers who may see the need to move between both words. Third, a unifying vision is laid out for a future where the advantages of both styles of data modeling can be shared in a single model.

DESIGN QUESTIONS

Any data designer who may be straddling the boundary between relational and non-relational database designs should consider the following set of questions.

What degree of normalization is sensible?

There is a wide range of possibilities with any given data set, as to how normalized or denormalized it can be. Taking our employment application example from above, we could, at one end of the spectrum, completely denormalize the data, putting it all into one entity where each record is, for example, an Applicant. Every "tuple" of this relation would have massive duplication of attributes, including information about the Positions, the questions, etc. On the other end of the spectrum, we could produce the

extremely normalized version of figure 11, and rely on join operations for even the simplest query.

The effective give and take of the normalization dichotomy is that normalization is worse for performance because it requires joins when disparate information is required together, whereas denormalization is more complex (because it may require more physical operations be done when changes occur) and more disk-heavy (because similar information may be stored multiple times).

Generally speaking, non-relational databases fall squarely on the denormalization side, because their distributed nature makes obtaining correlated information across multiple nodes difficult; when the schema model is more lax, there is little reason for developers to produce ultra-normalized designs in the first place.

Another way to ask this question is, “Should a relationship be embedded or referential?” Referential implies that the two entities are stored and accessed separately, whereas embedded implies (potential) denormalization [Murphy, 2009]. While the exact physical divisions for optimal performance are of course system-specific, there are some general terms that can distinguish between the two cases. If an object would be considered “first class” (that is, one of the important entities in the system)

Which entities participate in transactions together?

Are there distinct subsets of entities in the model where transactions involving multiple members of the subset are common, but transactions crossing subset boundaries are uncommon or nonexistent? This could point to a particular data layout, such as the Entity Groups concept in the Google App Engine Data Store. In our simple Employment Application example, there's a clear division between Positions and Applicants, in that it is uncommon that one would be seeking to update both the definition of the Position

itself in conjunction with one or more Applicant records. An alternative design where, for example, the Position record holds a pointer to the specific Applicant who was hired for the job, and the Applicant simultaneously changes a status column from "Applied" to "Hired" might complicate this situation, pointing to either the fact that the two entities should be within the same transaction group (potentially less performant or scalable) or that our data design is overcomplicated.

Alternate designs for transactions are also possible. Consider patterns such as the “Escrow Broker” pattern [Helland, 2007], where multiple parties all trust in one central actor to asynchronously commit (or roll back) a transaction. If the application design requires complex transactions spanning multiple entities, which may be physically distributed, the addition of such an abstraction can drastically simplify the process, rather than expecting the database infrastructure to simply handle it transparently.

Where are areas of high contention?

If the data store engine uses Optimistic Concurrency, as all most of the non-relational implementations we have considered do, then areas where very many simultaneous users might be updating the same entity should be avoided, or at least carefully considered, as they might precipitate locking problems and cause arbitrarily long wait times in user processes. As an example, consider a counter on a web page. If this counter was implemented as a single instance of an entity that multiple processes attempt to update every time the page is loaded, then high transactional throughput will cause potentially long waits for the page to be rendered--each process will attempt to read the current value of the counter and write a new value in a transaction, but that write will fail if any other transaction is already in progress, causing it to abort and retry. A more sane setup here might be to have each page access write a new record, for example into a

log table, and then have an offline process crawl the records of that table aggregating hit counts (which is the general paradigm of the map/reduce architecture).

What are the history requirements of the application?

Are there cases where it would be useful or necessary to view query results as they would have appeared at some point in history? For example, does an editable entity need to support a "revision history" property, or Undo operations? If so, engines that store all updated versions of a value (such as BigTable or Cassandra) may be the best choice, as this historical property can be exploited with no additional development.

Is Eventual Consistency an option?

There are certainly applications where eventual consistency is not adequate for the requirements of the system; for example, in a banking application, if there is a period of time where a billion dollar transaction appears to have only partially completed, it might be problematic. Less onerously, applications that depend on back and forth patterns of human interaction (say, instant messaging) do require that the system portray, at least locally, a consistent picture of the interaction, or else the participants may become confused.

Determining the exact tolerance for inconsistency of different portions of an application is a useful exercise, and decisions about these patterns should be documented along with the logical data models. Even for applications without plans for distributed operation, this kind of knowledge about the system can be used in system tuning; for example, if there is data that can tolerate some degree of inconsistency, that knowledge can be used to decrease observed latency by pushing a cache of data to the client, which

will potentially be temporarily inconsistent with the overall central data state of the system, but will seem "snappier" to users.

Practically, if eventual consistency is an option for the logical design of the system, then a physical strategy of *update queuing* may also be an option, where all writes to the database take place via non-blocking queue operations. This introduces additional latency, points of failure, and general complexity to the solution, but might be a suitable architecture in certain situations. Taking the concept even further, you might consider the entire system to be under an event-driven architecture, where all interactions between users and the persistent state of the system are enacted via asynchronous, non-blocking events or messages.

Alternately, if exact consistency is required, there may still be benefits of using eventual or weak consistency as a part of the indexing and lookup strategy. As described in [Taylor, 2009], it is possible to write the main entry of a piece of data atomically with full consistency, but then write index records without any atomicity guarantee. At that point, the application can be "mistrustful" of index entries, and always apply the same filters to both the index lookup and the data retrieval. This is a similar idea to Bloom filters, where the presence of a value is indicated (but not guaranteed) by a bit in a filter; the practices allows optimization of performance with respect to disk I/O, but no loss of correctness.

Does a Hash Table already model your problem?

There are certain problems that naturally point to Hash Table solutions – for example, dictionary models where a known key is the index to any given data set. If your data falls into this pattern, a non-relational database structure based on hash tables (including any of the key/value stores we've seen above) may be a good fit.

Is the Entity/Attribute/Value pattern inherent in the data?

If, as in our original example of Questions and Answers, your data naturally falls into a “Entity / Attribute / Value” pattern, then any number of non-relational databases may be a vastly better fit than a relational database, because of a basic mismatch in the structure of the data and the structures that relational databases and SQL queries provide. There are numerous examples of this in Biology, Artificial Intelligence, and the Semantic Web’s "Subject-Predicate-Object" triples in RDF.

Are there hierarchical or recursive relationships in the data?

While relational databases have adapted over the years to comfortably handle advanced tree or graph-like structures (e.g. the Nested Set model), if your data primarily exhibits such relationships, it is well worth examining the non-relational approaches presented here (especially the graph-oriented databases).

Are there natural functional boundaries to partition along?

Aside from horizontal scaling through homogenous distributed storage, as most of the non-relational database solutions do, there is another direction of parallelism that can be exploited: partitions between functional silos [Pritchett, 2008]. For example, data about products can be stored in one storage engine, whereas data about users can be stored in another.

Note that a partition between functional areas, if implemented as a physical database division (e.g. on multiple servers), is truly a boundary when it comes to data design. Using the running example above, if you put the database serving Positions on

one server, and the database serving Applicants on another server, the only way to produce a composite join of positions and applicants is by retrieving both separately and joining them manually, in memory. It is rare that any application will truly exhibit this level of separation naturally; even if there are areas whose functionality is completely disjoint, there will typically be some shared services, such as user identity management and access control, system constants, etc.

Another dimension to consider for partitioning, rather than functional areas per se, is for systems with disproportionate silos of data. For example, a photo sharing web site will have meta-data to run the system (users, groups, tags, etc.), and then will typically have two to three orders of magnitude more raw data in the actual photo assets it tracks. In this case, there is a clear partition between the two, and the scaling needs for both are quite different. In this case, running the meta-data on a traditional relational database, but running the large binary data on a distributed non-relational data store, might be a good option.

Are there compounding factors that might influence your design?

Though we give it only passing mention, it should also be obvious that pure logical design factors are not the only considerations when approaching the choice of database paradigm. [Brown, 2009] introduces a set of guidelines, including:

- Does the organization have licenses or funds for a commercial RDBMS?
- Does the current hardware setup of the organization support running an RDBMS?
- Does the application need to interact with or migrate other data, such as in legacy systems, where the relational paradigm is already in use?
- Does the organization have proper backup / restore / archival processes for relational databases?

- What is the skill set of the development team? (Otherwise known as, “Who wants to learn Erlang this weekend?”)
- What are your reporting requirements, in terms of both ad-hoc data queries and scheduled reports? Can you best satisfy them with a SQL interface?
- Do other systems need access to your data, and if so, is it via a SQL interface?

DESIGN STRATEGIES

This section offers a set of prescriptive tips and considerations to accompany the design process, in light of the relative merits of relational and non-relational databases.

Logical Model First

It is never a bad idea to spend the time during the upstream portion of a project to get a better understanding of the underlying purpose of the software. One of the best ways to do this is via a formal logical data design process. Whether it is done in UML, or sketched on a white board, an expert data modeler (relational or otherwise) will uncover an immense amount of knowledge about any non-trivial application by undergoing this type of effort.

Many of the problems data engineers face stem from the fact that we end up designing subconsciously to a particular physical model, rather than being able to work with the higher relational model. A standard principle we emerge with, then, is that it is always advisable to do **logical** data design first, regardless of the ultimate physical destination of the database

Ideally, there would be tools available to aid in this design and transition process. Unfortunately, such tools do not yet exist (or those that do are hopelessly outdated and

certainly not prepared for the advent of non-relational models). But in general, pen and paper sketches are far superior to doing nothing at all.

Consider Several Physical Approaches

Settling on a particular database technology should not necessarily be the end of the logical data design process. Many instances of poor relational database design could be fixed by considering a non-relational pattern, and vice versa. If you are set on modeling using a key/value store, consider writing a sketch of the problem in SQL first, possibly with an eye to avoiding certain SQL anti-patterns like the Entity / Attribute / Value problem explored above.

A master data designer should have familiarity with these different types of storage systems, for several reasons: to recognize that a problem she is designing for might actually be much more clearly expressed in another paradigm; to make correct design choices for systems that might start in one paradigm (probably a relational SQL database) and later migrate to another paradigm as scalability demands increase and functional fluidity decreases; and, on a meta level, to design future data storage engines that enable the "best of both worlds", changing in response to the many facets of the design process for an application.

Keep It Simple

Thought it may be a truism, it bears repeating: complexity itself is often the biggest enemy of any software design project. In that respect, a major goal of the logical design process should be to keep all things as simple as possible. Sometimes, that means that a simple relational database design—with tables, rows, columns, foreign keys, etc.—is the best and most familiar design. Other times, however, it may become apparent that forcing standard degrees of normalization on the data is costly and complex compared to

simply storing it in a semi-structured blob with a single key. There is no magic threshold for this kind of decision, and it certainly has something to do with the skill level of the development team in various areas. But it is a very worthwhile activity to keep an eagle eye watch on complexity in any project, and always strive for something simpler.

Play It Safe

If the logical design decision of using non-relational databases seems plausible for an application, but the choice of specific technology is daunting or unclear, one option is to build a simple non-relational layer on top of a relational database. This can be as lightweight as creating a simple key/value wrapper layer over a table in an existing relational database, such as MySQL, and then implementing the application in terms of that simple dictionary-like API. There are examples of this approach, such as the one used the company FriendFeed [Taylor, 2009]. Their concern was that in supporting high data volumes in standard MySQL tables, they encountered numerous operational problems around building and removing indices, etc. As an alternative, they revised the data model of FriendFeed to use a simple key/value implementation, with opaque values (which are actually compressed, serialized Python dictionaries created using zlib compression and Python's *pickle* serialization). The application layer then worked with the columns and values within these blobs, and created indices that were themselves database tables in MySQL, which could be created and removed efficiently as needed by the authors of the application.

Show Your True Consistency

If there are going to be areas within your application where consistency guarantees are relaxed, consider what patterns of user interaction design might best support this. For example, if transactions (such as credit card purchases) are not instantly

reflected in transaction summary views, a simple strategy such as labeling the view with "Current as of …" and a date can alleviate questions and worries. Users can be quite tolerant of temporary inconsistency if they are given enough information to understand its scope and resolution schedule.

The main pattern to avoid, in this area, is any case where the user might question if some action they took completed successfully. For example, if the option of uploading a photo might appear temporarily inconsistent in the overall photo view, the user might be tempted to upload it again, thus creating a duplicate. Feedback in such cases—such as a message stating, "Your photo has been uploaded, please wait up to 5 minutes for it to appear in this view"—is critical.

Stick To The Map (Reduce)

Map/Reduce is emerging as one of the most powerful tools in an analytical toolkit, and might have the power to conceptually supplant other paradigms for it (OLAP, Data Warehouses, Cubes, Star Schemas, etc.) Consider at the start what your analytics framework should be, and make allowances for it. For example, **Hive** is a system that does this type of work over a Hadoop map/reduce operation and exposes some querying primitives in a language called "QL", which is SQL-like, but also allows plugging in custom map reducers.

Evolve Gracefully

No schema stays the same forever. Regardless of the mode of a system's data interaction, it is important to create a plan for how future changes to the schema will be handled, ideally without a) requiring any downtime, or b) leaving a legacy code mess. Consider in your initial designs how this might occur, and it may lead to some allowances

in the original design, or in the choice of platform, that improve this picture down the line.

One potential development to support this would be for non-relational databases to explicitly track information about the "schema version" of stored data (as distinct from the data version), though the exact mechanism to do this in some cases is far from clear. [Strauss, 2009]

THE ONE TRUE DATABASE?

While the birth of cutting-edge non-relational databases is an exciting development in software, it is an unfortunate state of affairs that we, as engineers, must choose to move down one path or the other with our conceptual designs. Consider instead that eventually, relational and non-relational models might merge, or at least find some common ground. RDBMS vendors might begin offering a new type of service, in addition to (and well integrated with) their existing relational infrastructure, that emulates the behavior of these key/value stores, with minimal overhead. Non-relational entities could be another option, in addition to tables, that provide superior scalability and performance, offering a menu of additional services (transactions, locking access control, etc) that can be enable or disabled as desired (and permitted) by the performance requirements of the system. In many ways, this approach echoes the Microsoft philosophy (and indeed, looks similar to the Windows Azure offerings explored above).

On the other hand, there is much to be said for the Unix philosophy of having many small tools, each of which does a single job very well, and all of which interact through standard mechanisms. Rather than have one unified, all-things-to-all-people database management system, it could be that we are already on the right road, with a plethora of different tools, each geared to solve different problems well. Having a

healthy, competitive marketplace for such systems ensures that the systems that end up with high adoption will be those most battle tested and orthogonal with the actual needs of tomorrow's software systems.

Regardless of which of these directions one is a proponent of, there are a few key concepts that overlap both areas. This section explores several dimensions of that evolving relationship.

Modeling Constructs

While we may not end up with (or even desire) a single unifying database architecture, we can hope that another segment of the design space might become more unified: that of conceptual modeling. Use of UML has become widespread for object-oriented programming, but it is still a poor fit for data modeling. This is partly because, compared to object oriented designs, traditional relational designs are comparatively impoverished: there is no inheritance, no differentiated aggregation versus composition, no list types, etc. Mapping from a full UML design space down to a relational space is error prone and non-trivial[20]. More work should be done in this area, as there is potentially much to be gained from a consistent and transparent logical-to-physical mapping via tools.

Too, there are a range of logical characteristics that currently have no place in UML, but would be useful in designing the data models of tomorrow. For example, what is an entity's tolerance for inconsistency? Along what lines could it be partitioned? Are its relationships candidates for embedding or referencing?

Ultimately, a sound goal would be to achieve mathematical models for all types of relevant data model patterns that are as elegant and complete as the relational model itself.

[20] Use of Rational Rose to do this, in particular, is a painful and horrible experience. [Varley, 2009]

Schema Translation

Just as it would be ideal to have a modern, unified data modeling tool that could then transition into any number of Physical schema setups, it would be helpful to have adapter layers that could transform physical schemas from one database to another. A common use case for this might be to take an application that was built on a relational database platform and transition it to a non-relational store without any rewriting. Can we find a mechanical way to transform complex relational databases into key/value stores?

Theoretically, our tool kit could provide a "wizard" interface to translate from relational database schemas to non-relational schemas. As an input, a SQL schema is given, along with an indication of the target platform. The metadata from the SQL schema is then used to guide the user through a series of questions that disambiguate unknown cases and discern the user's design intent. The output depends on the platform.

For example, with a SQL to Google App Engine translator, the output might be a Python code file that defines the schema of the non-relational database, and additional code to enforce certain operations that were part of the relational database. Mappings might include:

- Each table becomes an entity; each column (except identity) become properties
- Identity columns are removed in favor of the key column, unless that identity column is intended to have business meaning
    - non-NULL columns are required ("required=True")
    - Columns with bound defaults get automatic values
    - Each SQL type would need to be mapped to a destination schema type varchar(1-500)->StringProperty; varchar(>500)->TextProperty; etc.
- Each foreign key becomes a reference

- Foreign key tables that do not contain other properties can be turned into "choices:" sets

Additionally, the interface would be several design aspects that would not be clear from the relational data design, but would have to be answered explicitly. For example

- For each single column primary key identity column, ask if it has "business import" or if it behind the scenes enough that it can be completely replaced.
- Integration with Google Accounts can replace any username / userid columns (created by, updated by, owner, etc.)
- Entity / Attribute / Value patterns could be identified and transformed into Expando properties
- In foreign key relationships, the user could describe both directions so that the proper names can be given to the references and back references. For example, an applicant has its Position reference, which is obvious, but a position will also have its Applicants reference, which returns a set of all the Applicants that refer to it.

Finally, there are some things that would require more research before they could properly be modeled:

- Are there any relational patterns that could be converted into multi-value properties? Perhaps look for one-to-many relationships with small numbers of values and no additional properties.
- Additional SQL capabilities, like GROUP BY, could be transformed to equivalent standard functions in a map-reduce paradigm, perhaps with intermediate storage or as materialized views.

- It would be a generally useful effort to craft templates for all of the known SQL functions in all of the non-relational paradigms – for example, a standard

Referential Overlays

Another useful tool in the new data modeling toolkit is the idea of a "referential overlay". The idea is that a conceptual layer could be developed between the logical schema and the current-version on-disk data, which has the ability to map ongoing access to the data through a virtual mutator [Strauss, 2009]. This could be a key part of any migration strategy between successive versions of the codebase, and might even be automatically produced.

Pluggable Architectures

Finally, one suggestion from [Moon, 2008] is that the distribution of a database might legitimately be dealt with separately from its model implementation, by using a sufficiently tiered architecture, where clients pass data requests to a library which does the intermediate work of partition lookup, data retrieval, read repair, relational re-mapping, etc. Under the covers, that library could potentially be dealing with a wide variety of different physical storage architectures, including both relational and non-relational database management systems. The trick is in making sure that the interface is sufficiently robust that the intent of the developer can be realized, while not being so complicated that it can hide subtle bugs or performance problems. There does not appear to be any consensus on a front-runner on this approach at present, but it is commendable that the development of non-relational distributed databases has spurred interest.

SECTION 7: ANALYSIS & CONCLUSIONS

> "Think of the Relational Model as being analogous to arithmetic, and the implementation as a calculator. The calculator could be an old, room-sized, gear and lever machine that takes minutes to produce a single answer. Does the clunkiness of such a calculator mean that arithmetic is "doomed"? [Bain, 2009]

This report has investigated the differences between traditional relational database modeling and several new forms of non-relational design that have arisen in response to the scaling challenges presented by modern web-scale software problems.

Can we now declare a winner in this battle? Far from it. We can, however, make some key observations about the differences.

At the core is the realization that relational database design is only one tool among many. Its supremacy in market share is well explained by its sound mathematical underpinnings, its general purpose data design framework, and the impressive engineering that has allowed it to perform at very high levels in most situations. But ultimately, it is not the hammer for all nails; its strengths and weakness are all the more visible in the light cast by a new breed of data management platforms. Generally, scalability is cited as the main reason to eschew relational databases for key/value stores; however, as we have shown in this report, there are a wide range of differences in expressive power: some in favor of relational databases and some in favor of key/value stores. There are also design decisions from an overall architecture point of view that favor one direction or the other.

Ultimately, as engineers, our goal should not be to merely settle for one paradigm or the other, but to envision a time when we can create databases that merge the strengths of both paradigms, with powerful abstractions that allow us to design our data in clear,

natural, concise ways, and then implement those designs in the most efficient way possible given the architectural constraints of the task at hand.

Non-relational databases don't allow us to express the types of designs we're "used to" in relational database modeling, but they can often give us equally good—and sometimes better—alternatives. In the best case, they encourage much simpler designs than relational databases do; in the worst case, they offer us no particular advantages, but offer avenues of scaling that cannot be achieved otherwise, and encourage alternative functional decompositions in our designs than we would have otherwise come up with.

Non-relational databases are a new breed of systems, built from the ground up with an entirely different goal from SQL and relational databases: rather than pouring development effort into building abstraction layers on top of the raw storage to allow hapless developers to get near-optimal results regardless of how clumsy their schemas and queries are, this new set of tools requires first and foremost that scalability and efficiency are king, and that any operations built on top of those primitives must be created with care and significant engineering investment.

Along those lines, one general way to state the advantages of using non-relational databases is that they put the developer closer to the machine - more in charge of the specific operations that are done to structure, persist, and fetch data. As has been shown over the history of computing, the point of optimal closeness to the machine is under continual metamorphosis. Very few programmers today write in directly assembly language, in part because computers have gotten faster, but also in part because we have, over time, learned to create and use abstractions that cleanly and efficiently implement our intended functionality in terms of the machine. Few programmers could even write assembly code that is as optimized and efficient as modern compilers do when given

some high level language code to compile; the abstractions are themselves the product of many years of world-class research and engineering.

So it is with SQL databases. Even a giant tome like [Garcia-Mollina, 2008] can only touch on many areas of engineering and research that make today's commercial databases as fast and efficient as they are. In some sense, the development of relational databases itself is ahead of its time; the fact that a pristine mathematical model of relations is today the primary way in which programmers design and interact with data is something of a miracle of engineering. The very key to their ability to do this, however, is that the model—relational data design and SQL—is a time tested, mathematically grounded abstraction layer. It is not perfect, but neither is it outdated or useless.

This author would advocate, therefore, that the developments exemplified by non-relational databases should not remain an outside challenger to the legacy of relational databases, but should instead be researched, understood, and eventually, incorporated into the relational model. There's nothing to say that implementation as a key/value store shouldn't be part of the suite of implementation choices for a database whose data is structured relationally; likewise, there is room in the world of relational databases for the conceptual data design advantages offered by non-relational databases. The option to use optimistic concurrency control; to keep multiple versions of a cell per the columnar database model; to accept and support semi-structured (or run-time structured) data efficiently; to maintain multiple simultaneous values for a cell; and to scale across a cluster using some sort of ancestry or grouping relationship—these would all be conceptually coherent additions to the relational database world, provided the mathematical model for their incorporation is sound, and the configuration of the options is transparent and cohesive.

Bibliography

Abadi, D. "Data Management in the Cloud: Limitations and Opportunities". IEEE Computer Society, Bulletin of the Technical Committee on Data Engineering, January 0000 Vol. 0 No. 0.

Aiyer, A; Anderson, E; Li, X; Shah, M; Wylie, J. "Consistability: Describing usually consistent systems." Proceedings of HotDep 2008, The 4th workshop on Hot Topics in Dependability (7 Dec 2008), San Diego, California, USA.

Amazon.com. “Amazon SimpleDB Product Home Page”. Accessed in July, 2009. http://aws.amazon.com/simpledb/

Baeza-Yates, R; Ramakrishnan, R. "Data Challenges at Yahoo!" In Proceedings of the 11th international Conference on Extending Database Technology: Advances in Database Technology (Nantes, France, March 25 - 29, 2008). EDBT '08, vol. 261. ACM, New York, NY, 652-655.

Bain, Tony. "Is the Relational Database Doomed?". ReadWrite Enterprise. Feb 12 2009. http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php?p=3

Barreto, C. “NoSQL: leading Cloud's ‘NoBah’ movement?”. Digital Walkabout, July 2009. http://charltonb.typepad.com/weblog/2009/07/nosql-leading-clouds-nobah-movement.html

Bernstein, P. A. and Goodman, N. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (Jun. 1981), 185-221. DOI= http://doi.acm.org/10.1145/356842.356846

Brantner, M; Florescu, D; Graf, D; Kossmann, D; Kraska, T. "Building a database on S3". Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 251–264. 2008.

Brown, S. “To SQL or not to SQL?”. CodingTheArchitecture.com, July 2009. http://www.codingthearchitecture.com/2009/07/21/to_sql_or_not_to_sql.html

Cates, Josh. Robust and Efficient Data Management for a Distributed Hash Table. Master's thesis, Massachusetts Institute of Technology, May 2003.

Chang, F; Dean, J; Ghemawat, S; Hsieh, W; Wallach, D; Burrows, M; Chandra, T; Fikes, A. "Bigtable: A Distributed Storage System for Structured Data", Seventh Symposium on Operating System Design and Implementation. (2006)

Codd, E. “Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks”. IBM, San Jose, California, IBM Research Report RJ599, June 1969.

Cooper, B; Ramakrishnan, Raghu; Srivastava, Utkarsh; Silberstein, Adam; Bohannon, Philip; Jacobsen, Hans-Arno; Puz, Nick; Weaver, Daniel; Yerneni, Ramana. "PNUTS: Yahoo!'s hosted data serving platform". Proceedings of the VLDB Endowment, SESSION: Industrial, application, and experience sessions: massive data, pp 1277-1288 (2008)

Das, S; Agrawal, D; Abbadi, A. "ElasTraS: An Elastic Transactional Data Store in the Cloud". USENIX 2009 HotCloud Conference Proceedings. (2009)

Das, S; Antony, S; Agrawal, D; Abbadi, A. "Clouded Data: Comprehending Scalable Data Management Systems". UCSB Computer Science Technical Report 2008-18, November 2008.

DeCandia, G; Hastorun, D; Jampani, M; Kakulapati, G; Lakshman, A; Pilchin, A; Sivasubramanian, A; Vosshall, P; Vogels, W. "Dynamo: Amazon's Highly Available Key-Value Store", in the Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, October 2007.

Dinu, Valentin; Nadkarni, Prakash. "Guidelines for the effective use of entity-attribute-value modeling for biomedical databases". Int Journal of Medical Informatics 76 (11-12): 769-779. (December 2007) http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=17098467&query_hl=2&itool=pubmed_docsum.

Garcia-Mollina, H; Ullman, J; Widom, J. Database Systems: The Complete Book, 2nd Edition. Prentice-Hall, June 2008.

Geambasu, R; Gribble, S; Levy, H. "CloudViews: Communal Data Sharing in Public Clouds". USENIX 2009 HotCloud Conference Proceedings. (2009) http://www.usenix.org/event/hotcloud09/tech/full_papers/geambasu.pdf

Gelernter, D. “Generative communication in Linda”. ACM Transactions on Programming Languages and Systems, volume 7, number 1, January 1985

Gilbert, S; Lynch, N. "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services". ACM SIGACT News, 2002.

Google. “GQL Reference”, Google App Engine Documentations, 2009. http://code.google.com/appengine/docs/python/datastore/gqlreference.html

Hay, David. Data Model Patterns. Dorset House Publishing Company, Incorporated (1995)

Helland, P. “Life beyond Distributed Transactions: an Apostate’s Opinion”. 3rd Biennial Conference on Innovative DataSystems Research (CIDR), 2007.

January 7-10, Asilomar, California USA.

Hsieh, W; Madhavan, Jayant; Pike , Rob. “Data management projects at Google”. SIGMOD Conference, 2006, pp. 725-726.

Hui, M; Jiang, W; Li, G; Zhou, Y. "Supporting Database Applications as a Service". ICDE, 2009. http://www.comp.nus.edu.sg/~huimei/papers/multi.pdf

Jennings, R. "Retire Your Data Center”. Visual Studio Magazine, February 2008.

Kamfonas, M. "Recursive Hierarchies: The Relational Taboo!". The Relational Journal. October/November, 1992. http://www.kamfonas.com/id3.html

Kreps, J. Project Voldemort presentation at June 2009 NoSQL Meetup, San Francisco. http://static.last.fm/johan/nosql-20090611/voldemort_nosql.pdf

Lakshman, A. Cassandra Project Presentation at June 2009 NoSQL Meetup, San Francisco. http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf

Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM. Volume 21, Number 7. July 1978.

Moon, C. Dynomite Project Presentation at June 2009 NoSQL Meetup, San Francisco. http://static.last.fm/johan/nosql-20090611/dynomite_nosql.pdf

Murphy, R; Merriman, D. “MongoDB Schema Design”. MongoDB.org, June 2009. http://www.mongodb.org/display/DOCS/Schema+Design

Murty, James. Programming Amazon Web Services, First edition. O'Reilly, 2008. http://portal.acm.org.ezproxy.lib.utexas.edu/citation.cfm?id=1407893

MySQL. “MySQL 5.0 Reference Manual”. 2009. http://dev.mysql.com/doc/refman/5.0/en/index.html

Olston, C; Reedy, B; Srivastavaz, U; Kumarx, R; Tomkins, A. “Pig Latin: A Not-So-Foreign Language for Data Processing”. SIGMOD’08, June 9–12, 2008, Vancouver, BC, Canada. http://www.cs.cmu.edu/~olston/publications/sigmod08.pdf

Pritchett, D. “BASE: An Acid Alternative”. ACM Queue, vol. 6, no. 3, July 2008. http://queue.acm.org/detail.cfm?id=1394128

Stonebraker, M. “The Case for Shared Nothing Architecture”. Database Engineering, Volume 9, Number 1 (1986). http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf

Stonebraker, M, et al.. "C-Store: A Column-oriented DBMS," Proc 2005 VLDB Conference, Trondheim, Norway, Sept. 2005.

Stonebraker, M. "The End of a DBMS Era (Might be Upon Us)". Blogs at the Communications of the ACM, http://cacm.acm.org/blogs/blog-cacm/32212-the-end-of-a-dbms-era-might-be-upon-us/fulltext

Stonebraker, M. and Cetintemel, U. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st international Conference on Data Engineering* (April 05 - 08, 2005). ICDE. IEEE Computer Society, Washington, DC, 2-11. DOI= http://dx.doi.org.ezproxy.lib.utexas.edu/10.1109/ICDE.2005.1

Strauss, D. “Giving schema back its good name”. FourKitchens Blog, July 2009. http://fourkitchens.com/blog/2009/07/05/how-schema-got-bad-name

Taylor, B. “How FriendFeed uses MySQL to store schema-less data.” February, 2009. http://bret.appspot.com/entry/how-friendfeed-uses-mysql

Vogels, Werner. "Eventually Consistent". Communications of the ACM, Volume 52, Issue 1 (January 2009).

Yu, Y; Isard, M; Fetterly, D; Budiu, M; Erlingsson, U; Gunda, P. K.; Currey, J. “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language”. Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December 8-10, 2008.

Wei, Z; Dejun, J; Pierre, G; Chi, C; van Steen, M. "Service-Oriented Data Denormalization for Scalable Web Applications". ACM 978-1-60558-085-2/08/04. WWW 2008 / Refereed Track: Performance and Scalability April 21-25, 2008. Beijing, China (2008)[Campbell, W. G. 1990. Form and Style in Thesis Writing, a Manual of Style. Chicago: The University of Chicago Press.

Vita

Ian Varley was born in Albany, NY in 1975, to Candice and Thomas Varley. He attended Skidmore College in Saratoga Springs, NY, where he graduated summa cum laude in 1997 with a Bachelor of Arts in both Music and Philosophy. Since 1998, he has worked as a Software Engineer and Database Architect for companies in San Francisco, Houston, and Austin. He has also taught 6^{th} – 8^{th} grade Computer Science. He is an avid musician, and tours the world regularly playing with various bands (most recently, and stressfully, during the completion of this report).

Permanent address: 4221 Mattie St., Austin, TX 78723

This report was typed by the author.