NoSQL and Big Data for the Oracle Professional

Iggy Fernandez, NoCOUG

In this paper, I will dissect and demystify NoSQL technology. The relational camp derides NoSQL technology because NoSQL technology does not play by the rules of the relational camp. Therefore the relational camp is ignoring the opportunity to incorporate the innovations of the NoSQL camp into mainstream database management systems. For its part, the NoSQL camp derides the relational model as unable to satisfy the performance, scalability, and availability needs of today. I claim that the NoSQL camp derides the relational model because it does not sufficiently understand it. I will go so far as to claim that the NoSQL camp does not fully understand its own innovations; it believes that they are incompatible with the relational model and it therefore does not see the opportunity to *strengthen* the relational model. A very strong assertion which I will defend as I go along.

Disruptive Innovation

NoSQL technology is a "disruptive innovation" in the sense used by Harvard professor Clayton M. Christensen. In *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*, Professor Christensen defines disruptive innovations and explains why it is dangerous to ignore them:

> *"Generally, disruptive innovations were technologically straightforward, consisting of off-the-shelf components put together in a product architecture that was often simpler than prior approaches. They offered less of what customers in established markets wanted and so could rarely be initially employed there. They offered a different package of attributes valued only in emerging markets remote from, and unimportant to, the mainstream."*

Established players usually ignore disruptive innovations because they do not see them as a threat to their bottom lines. In fact, they are more than happy to abandon the low-margin segments of the market and their profitability actually increases when they do so. The disruptive technologies eventually take over most of the market.

An example of a disruptive innovation is the personal computer. The personal computer was initially targeted only at the home computing segment of the market. Established manufacturers of mainframe computers and minicomputers did not see PC technology as a threat to their bottom lines. Eventually, however, PC technology came to dominate the market and established computer manufacturers such as Digital Equipment Corporation, Prime, Wang, Nixdorf, Apollo, and Silicon Graphics went out of business.

So where lies the dilemma? Professor Christensen explains:

> *"In every company, every day, every year, people are going into senior management, knocking on the door saying: 'I got a new product for us.' And some of those entail making better products that you can sell for higher prices to your best customers. A disruptive innovation generally has to cause you to go after new markets, people who aren't your customers. And the product that you want to sell them is something that is just so much more affordable and simple that your current customers can't buy it. And so the choice that you have to make is: Should we make better products that we can sell for better profits to our best customers. Or maybe we ought to make worse products that none of our customers would buy that would ruin our margins. What should we do? And that really is the dilemma."*

Exactly in the manner that Christensen described, the e-commerce pioneer Amazon.com created an in-house product called Dynamo in 2007 to meet the performance, scalability, and availability needs of its own e-commerce platform after it concluded that mainstream database management systems were not capable of satisfying those needs. The most notable aspect of Dynamo was the apparent break with the relational model; there was no mention of relations, relational algebra, or SQL.

I recommend that you take the time to listen to this five-minute YouTube video by Professor Christensen before reading the remainder of the series.

Requirements and Assumptions

As I mentioned, the NoSQL movement got its big boost from the e-commerce giant Amazon. Amazon started out by using Oracle Database for its e-commerce platform but later switched to a proprietary database management system called Dynamo that it built in-house. Dynamo is the archetypal NoSQL product; it embodies all the innovations of the NoSQL camp. The Dynamo requirements and assumptions are documented in the paper Dynamo: Amazon's Highly Available Key-value Store published in 2007. Here are some excerpts from that paper:

> *"Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers."*
>
> *"There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications."*
>
> *"Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability."*
>
> *"Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability."*
>
> *"… since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts."*

To paraphrase, Amazon's requirements were *extreme* performance, *extreme* scalability, and *extreme* availability, surpassing anything that had ever been achieved before. Also, Amazon's prior experience with the relational model led it to conclude that the only way to satisfy these requirements was to stop playing by the rules of the relational camp. If you belong in the relational camp, please suspend disbelief while I explain how Amazon achieved its ends. You will be in a better position to pass judgment on NoSQL technology once you understand each Amazon innovation.

Functional Segmentation

Amazon's *pivotal* design decision was to break its monolithic enterprise-wide database service into simpler component services such as a best-seller list service, a shopping cart service, a customer preferences service, a sales rank service, and a product catalog service. This avoided a single point of failure. In an interview for the *NoCOUG Journal*, Amazon's first database administrator, Jeremiah Wilton explains the rationale behind Amazon's approach:

> *"The best availability in the industry comes from application software that is predicated upon a surprising assumption: The databases upon which the software relies will inevitably fail. The better the software's ability to continue operating in such a situation, the higher the overall service's availability will be. But isn't Oracle unbreakable? At the database level, regardless of the measures taken to improve availability, outages will occur from time to time. An outage may be from a required upgrade or a bug. Knowing this, if you engineer application software to handle this eventuality, then a database outage will have less or no impact on end users. In summary, there are many ways to improve a single database's availability. But the highest availability comes from thoughtful engineering of the entire application architecture."*

As an example, the shopping cart service should not be affected if the checkout service is unavailable or not performing well.

I said that this was the *pivotal* design decision made by Amazon. I cannot emphasize this enough. If you resist functional segmentation, you are *not* ready for NoSQL. If you miss the point, you will not understand NoSQL.

Note that functional segmentation results in simple hierarchical schemas. Here is an example of a simple hierarchical schema from Ted Codd's 1970 paper on the relational model, meticulously reproduced in the 100th issue of the *NoCOUG Journal*. This schema stores information about employees, their children, their job histories, and their salary histories.

```
employee (man#, name, birthdate)
children (man#, childname, birthyear)
jobhistory (man#, jobdate, title)
salaryhistory (man#, jobdate, salarydate, salary)
```

Functional segmentation is the underpinning of NoSQL technology but it does not present a conflict with the relational model; it is simply a physical database design decision. Each functional segment is usually assigned its own

standalone database. The collection of functional segments could be regarded as a single distributed database. However, distributed transactions are verboten in the NoSQL world. Functional segmentation can therefore result in temporary inconsistencies if, for example, the shopping cart data is not in the same database as the product catalog and occasional inconsistencies result. Occasionally, an item that is present in a shopping cart may go out of stock. Occasionally, an item that is present in a shopping cart may be repriced. The problems can be resolved when the customer decides to check out, if not earlier. As an Amazon customer, I occasionally leave items in my shopping cart but don't complete a purchase. When I resume shopping, I sometimes get a notification that an item in my shopping chart is no longer in stock or has been repriced. This technique is called "eventual consistency" and the application is responsible for ensuring that inconsistencies are eventually corrected. Randy Shoup, one of the architects of eBay's ecommerce platform, explains how:

> *"At eBay, we allow absolutely no client-side or distributed transactions of any kind – no two-phase commit. In certain well-defined situations, we will combine multiple statements on a single database into a single transactional operation. For the most part, however, individual statements are auto-committed. While this intentional relaxation of orthodox ACID properties does not guarantee immediate consistency everywhere, the reality is that most systems are available the vast majority of the time. Of course, we do employ various techniques to help the system reach eventual consistency: careful ordering of database operations, asynchronous recovery events, and reconciliation or settlement batches. We choose the technique according to the consistency demands of the particular use case."* (Scalability Best Practices: Lessons from eBay)

The eventual consistency technique receives a lot of attention because it is supposedly in conflict with the relational model. We will return to this subject later in this series and argue that eventual consistency is *not* in conflict with the relational model.

Sharding

Amazon's next design decision was "sharding" or horizontal partitioning of all the tables in a hierarchical schema. Hash-partitioning is typically used. Each table is partitioned in the same way as the other tables in the schema and each set of partitions is placed in a separate database referred to as a "shard." The shards are independent of each other; that is, there is no clustering (as in Oracle RAC) or federation (as in IBM DB2).

Note that the hierarchical schemas that result from functional segmentation are always shardable; that is, hierarchical schemas are shardable by definition.

Returning to the example from Ted Codd's 1970 paper on the relational model:

```
employee (man#, name, birthdate) with primary key (man#)
children (man#, childname, birthyear) with primary key (man#, childname)
jobhistory (man#, jobdate, title) with primary key (man#, jobdate)
salaryhistory (man#, jobdate, salarydate, salary) with primary key (man#, jobdate, salarydate)
```

Note that the jobhistory, salaryhistory, and children tables have composite keys. In each case, the leading column of the composite key is the man#. Therefore, all four tables can be partitioned using the man#.

Sharding is an essential component of NoSQL designs but it does not present a conflict with the relational model; it too is simply a physical database design decision. In the relational model, the collection of standalone databases or shards can be logically viewed as a single distributed database.

Replication and Eventual Consistency

By now, you must be wondering when I'm going to get around to explaining how to create a NoSQL database. When I was a junior programmer, quite early in my career, my friends and I were assigned to work on a big software development project for which we would have to use technologies with which we were completely unfamiliar. We were promised that training would be provided before the project started. The very first thing the instructor said was (paraphrasing) "First you have to insert your definitions into the C.D.D." and he walked to the board and wrote the commands that we needed for the purpose. Needless to say, we were quite flustered because we had no idea what those "definitions" might be or what a "C.D.D." was and how it fit into the big picture.

NoSQL is being taught without reference to the big picture. None of the current books on NoSQL mention functional segmentation even though it is the underpinning principle of NoSQL. All the current books on NoSQL imply that NoSQL principles are in conflict with the relational model. If you are in a hurry to create your first NoSQL database, I can recommend Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. But as one of the

world's greatest geniuses Leonardo da Vinci has said: *"Those who are in love with practice without science are like the sailor who gets into a ship without rudder or compass, who is never certain where he is going. Practice must always be built on sound theory … The painter who copies by practice and judgement of eye, without rules, is like a mirror which imitates within itself all the things placed before it without any understanding of them."* (On the errors of those who rely on practice without science).

Continuing the train of thought from the previous section, the Dynamo developers saw that one of the keys to extreme availability was data replication. Multiple copies of the shopping cart are allowed to exist and, if one of the replicas becomes unresponsive, the data can be served by one of the other replicas. However, because of network latencies, the copies may occasionally get out of sync and the customer may occasionally encounter a stale version of the shopping cart. Once again, this can be handled appropriately by the application tier; the node that falls behind can catch up eventually or inconsistencies can be detected and resolved at an opportune time, such as at checkout. This technique is called "eventual consistency."

The inventor of relational theory, Dr. Codd, was acutely aware of the potential overhead of consistency checking. In his 1970 paper, he said:

> *"There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs.* ***Naturally, such checking will slow these operations down.*** *[emphasis added] If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently."*

In other words, the inventor of relational theory would not have found a conflict between his relational model and the "eventual consistency" that is one of the hallmarks of the NoSQL products of today. However, the Dynamo developers imagined a conflict because it quite understandably conflated the relational model with the ACID guarantees of database management systems. However, ACID has *nothing* to do with the relational model per se (although relational theory does come in very handy in defining consistency constraints); pre-relational database management systems such as IMS provided ACID guarantees and so did post-relational object-oriented database management systems.

I should not defend eventual consistency simply by using a convenient quote from the writings of Dr. Codd. *"The devil can cite Scripture for his purpose. An evil soul producing holy witness is like a villain with a smiling cheek, a goodly apple rotten at the heart. O, what a goodly outside falsehood hath!"* (from the Shakespeare play The Merchant of Venice) If I am in favor of eventual consistency, I should explain *why*, not simply quote from the writings of Dr. Codd. If I can defend my own beliefs, I free myself to disagree with Dr. Codd as I plan to do later in this series. I have in fact come to accept that real-time consistency checking should be a design *choice* not a scriptural mandate. I may have had a different opinion in the past but *"a foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. … Speak what you think now in hard words, and to-morrow speak what to-morrow thinks in hard words again, though it contradict every thing you said to-day."* (from the Emerson essay Self-Reliance).

The tradeoff between consistency and performance is as important in the wired world of today as it was in Dr. Codd's world. We cannot cast stones at Dynamo for the infraction of not guaranteeing the synchronization of replicated data (or allowing temporary inconsistencies between functional segments), because violations of the consistency requirement are equally commonplace in the relational camp. The replication technique used by Dynamo has a close parallel in the technique of "multimaster replication" used in the relational camp. Application developers in the relational camp are warned about the negative impact of integrity constraints.[1,2,3,4] And, most importantly, no mainstream DBMS currently implements the SQL-92 "CREATE ASSERTION" feature that is necessary to provide the consistency guarantee. For a detailed analysis of this

---

[1] *"Using primary and foreign keys can impact performance. Avoid using them when possible."* (http://docs.oracle.com/cd/E17904_01/core.1111/e10108/adapters.htm#BABCCCIH)

[2] *"For performance reasons, the Oracle BPEL Process Manager, Oracle Mediator, human workflow, Oracle B2B, SOA Infrastructure, and Oracle BPM Suite schemas have no foreign key constraints to enforce integrity."* (http://docs.oracle.com/cd/E23943_01/admin.1111/e10226/soaadmin_partition.htm#CJHCJIJI)

[3] *"For database independence, applications typically do not store the primary key-foreign key relationships in the database itself; rather, the relationships are enforced in the application."* (http://docs.oracle.com/cd/E25178_01/fusionapps.1111/e14496/securing.htm#CHDDGFHH)

[4] *"The ETL process commonly verifies that certain constraints are true. For example, it can validate all of the foreign keys in the data coming into the fact table. This means that you can trust it to provide clean data, instead of implementing constraints in the data warehouse."* (http://docs.oracle.com/cd/E24693_01/server.11203/e16579/constra.htm#i1006300)

anomaly, refer to Toon Koppelaars's article "CREATE ASSERTION: The Impossible Dream" in the August 2013 issue of the *NoCOUG Journal*.

The False Premise of NoSQL

The final hurdle was extreme performance, and that's where the Dynamo developers went astray. The Dynamo developers believed that the relational model imposes a "join penalty" and therefore chose to store data as "blobs." This objection to the relational model is colorfully summarized by the following statement misattributed to Esther Dyson, the editor of the Release 1.0 newsletter, *"Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car."* [5] The statement dates back to 1988 and was much quoted when object-oriented databases were in vogue.

Since the shopping cart is an object, doesn't disassembling it for storage make subsequent data retrieval and updates inefficient? The belief stems from an unfounded assumption that has found its way into every mainstream DBMS—that every table should map to physical storage. In reality, the relational model is a logical model and, therefore, it does not concern itself with storage details at all. It would be perfectly legitimate to store the shopping cart in a physical form that resembled a shopping cart while still offering a relational model of the data complete with SQL. In other words, the physical representation could be optimized for the most important use case—retrieving the entire shopping-cart object using its key—without affecting the relational model of the data. It would also be perfectly legitimate to provide a non-relational API for the important use cases. Dr. Codd himself gave conditional blessing to such non-relational APIs in his 1985 Computerworld article, "Is Your DBMS Really Relational?", in which he says, *"If a relational system has a low-level (single-record-at-a-time) language, that low level [should not] be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time)."*

The key-blob or "key-value" approach used by Dynamo and successor products would be called "zeroth" normal form in relational jargon. In his 1970 paper, Dr. Codd says: *"Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on. For example, one of the domains on which the relation employee is defined might be salary history. An element of the salary history domain is a binary relation defined on the domain date and the domain salary. The salary history domain is the set of all such binary relations. At any instant of time there are as many instances of the salary history relation in the data bank as there are employees. In contrast, there is only one instance of the employee relation."* In common parlance, a relation with non-simple domains is said to be in zeroth normal form or unnormalized. Dr. Codd suggested that unnormalized relations should be normalized for ease of use. Here again is the unnormalized employee relation from Dr. Codd's paper:

```
employee (
  employee#,
  name,
  birthdate,
  jobhistory (jobdate, title, salaryhistory (salarydate, salary)),
  children (childname, birthyear)
)
```

The above unnormalized relation can be decomposed into four normalized relations as follows.

```
employee' (employee#, name, birthdate)
jobhistory' (employee#, jobdate, title)
salaryhistory' (employee#, jobdate, salarydate, salary)
children' (employee#, childname, birthyear)
```

However, this is *not* to suggest that these normalized relations must necessarily be mapped to individual buckets of physical storage. Dr. Codd differentiated between the stored set, the named set, and the expressible set. In the above example, we have one unnormalized relation and four normalized relations, if we preferred it, the unnormalized employee relation could be the only member of the stored set. Alternatively, if we preferred it, *all* five relations could be part of the stored set; that is, we could legitimately store redundant representations of the data. However, the common belief blessed by current practice is that the normalized relations should be the only members of the stored set.

---

[5] The statement cannot be found in the Release 1.0 archives at http://www.sbw.org/release1.0/. However, the following statement appears in the September 1989 issue of Release 1.0: *"You can keep a car in a file cabinet because you can file the engine components in files in one drawer, and the axles and things in another, and keep a list of how everything fits together. You can, but you wouldn't want to."* (http://downloads.oreilly.com/radar/r1/09-89.pdf). Somewhere down the line, the original text must have been paraphrased and the paraphrased text attributed to Esther Dyson instead of the original text. Thanks to author Akmal Chaudhri for sending me the right source.

Even if the stored set contains only normalized relations, they need not map to different buckets of physical storage. Oracle is unique among mainstream database management systems in providing a convenient construct called the "table cluster" that is suitable for hierarchical schemas. In Dr. Codd's example, employee# would be the cluster key, and rows corresponding to the same cluster key from all four tables could be stored in the same physical block on disk thus avoiding the join penalty. If the cluster was a "hash cluster," no indexes would be required for the use case of retrieving records belonging to a single cluster key.

The mistake made by the Dynamo developers is really a mistake perpetuated by the relational camp but it is a mistake nevertheless.

Table Clusters in Oracle Database—Demonstration

Here's a demonstration of using Oracle table clusters to store records from four tables in the same block and retrieving all the components of the "employee cart" without using indexes. First we create four normalized tables and prove that all the records of a single employee including job history, salary history, and children are stored in a *single* database block so that there is never any join-penalty when assembling employee data. Then we create an object-relational view that assembles employee information into a single unnormalized structure and show how to insert into this view using an "INSTEAD OF" trigger.

The following demonstration was performed using Oracle Database is 11.2.0.2.

```
SQL*Plus: Release 11.2.0.2.0 Production on Sun Jul 28 19:44:23 2013

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
```

First, we create a table cluster and add four tables to the cluster.

```
SQL> CREATE CLUSTER employees (employee# INTEGER) hashkeys 1000;

Cluster created.

SQL> CREATE TABLE employees
  2  (
  3    employee# INTEGER NOT NULL,
  4    name VARCHAR2(16),
  5    birth_date DATE,
  6    CONSTRAINT employees_pk PRIMARY KEY (employee#)
  7  )
  8  CLUSTER employees (employee#);

Table created.

SQL> CREATE TABLE job_history
  2  (
  3    employee# INTEGER NOT NULL,
  4    job_date DATE NOT NULL,
  5    title VARCHAR2(16),
  6    CONSTRAINT job_history_pk PRIMARY KEY (employee#, job_date),
  7    CONSTRAINT job_history_fk1 FOREIGN KEY (employee#) REFERENCES employees
  8  )
  9  CLUSTER employees (employee#);

Table created.

SQL> CREATE TABLE salary_history
  2  (
  3    employee# INTEGER NOT NULL,
  4    job_date DATE NOT NULL,
  5    salary_date DATE NOT NULL,
  6    salary NUMBER,
  7    CONSTRAINT salary_history_pk PRIMARY KEY (employee#, job_date, salary_date),
  8    CONSTRAINT salary_history_fk1 FOREIGN KEY (employee#) REFERENCES employees,
  9    CONSTRAINT salary_history_fk2 FOREIGN KEY (employee#, job_date) REFERENCES job_history
 10  )
 11  CLUSTER employees (employee#);
```

```
Table created.

SQL> CREATE TABLE children
  2  (
  3    employee# INTEGER NOT NULL,
  4    child_name VARCHAR2(16) NOT NULL,
  5    birth_date DATE,
  6    CONSTRAINT children_pk PRIMARY KEY (employee#, child_name),
  7    CONSTRAINT children_fk1 FOREIGN KEY (employee#) REFERENCES employees
  8  )
  9  CLUSTER employees (employee#);

Table created.
```

Then we insert data into all four tables. We find that all the records have been stored in the same database block even though they belong to different tables. Therefore the join-penalty has been eliminated.

```
SQL> INSERT INTO employees VALUES (1, 'IGNATIUS', '01-JAN-1970');

1 row created.

SQL> INSERT INTO children VALUES (1, 'INIGA', '01-JAN-2001');

1 row created.

SQL> INSERT INTO children VALUES (1, 'INIGO', '01-JAN-2002');

1 row created.

SQL> INSERT INTO job_history VALUES (1, '01-JAN-1991', 'PROGRAMMER');

1 row created.

SQL> INSERT INTO job_history VALUES (1, '01-JAN-1992', 'DATABASE ADMIN');

1 row created.

SQL> INSERT INTO salary_history VALUES (1, '01-JAN-1991', '1-FEB-1991', 1000);

1 row created.

SQL> INSERT INTO salary_history VALUES (1, '01-JAN-1991', '1-MAR-1991', 1000);

1 row created.

SQL> INSERT INTO salary_history VALUES (1, '01-JAN-1992', '1-FEB-1992', 2000);

1 row created.

SQL> INSERT INTO salary_history VALUES (1, '01-JAN-1992', '1-MAR-1992', 2000);

1 row created.

SQL> SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) AS block_number FROM employees where
employee# = 1;

BLOCK_NUMBER
------------
       22881

SQL> SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) AS block_number FROM children where
employee# = 1;

BLOCK_NUMBER
------------
       22881

SQL> SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) AS block_number FROM job_history where
employee# = 1;

BLOCK_NUMBER
```

```
------------
       22881

SQL> SELECT DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) AS block_number FROM salary_history where
employee# = 1;

BLOCK_NUMBER
------------
       22881
```

Next we create an object-relational view that presents each employee as an object.

```
SQL> CREATE OR REPLACE TYPE children_rec AS OBJECT (child_name VARCHAR2(16), birth_date DATE)
  2  /

Type created.

SQL> CREATE OR REPLACE TYPE children_tab AS TABLE OF children_rec
  2  /

Type created.

SQL> CREATE OR REPLACE TYPE salary_history_rec AS OBJECT (salary_date DATE, salary NUMBER)
  2  /

Type created.

SQL> CREATE OR REPLACE TYPE salary_history_tab AS TABLE OF salary_history_rec
  2  /

Type created.

SQL> CREATE OR REPLACE TYPE job_history_rec AS OBJECT (job_date DATE, title VARCHAR2(16),
salary_history SALARY_HISTORY_TAB)
  2  /

Type created.

SQL> CREATE OR REPLACE TYPE job_history_tab AS TABLE of job_history_rec
  2  /

Type created.

SQL> create or replace view employees_view as
  2  SELECT
  3    employee#,
  4    name,
  5    birth_date,
  6    CAST
  7    (
  8      MULTISET
  9      (
 10        SELECT
 11          child_name,
 12          birth_date
 13        FROM children
 14        WHERE employee#=e.employee#
 15      )
 16      AS children_tab
 17    ) children,
 18    CAST
 19    (
 20      MULTISET
 21      (
 22        SELECT
 23          job_date,
 24          title,
 25          CAST
 26          (
 27            MULTISET
 28            (
 29              SELECT salary_date, salary
```

```
 30              FROM salary_history
 31              WHERE employee#=e.employee#
 32              AND job_date=jh.job_date
 33            )
 34            AS salary_history_tab
 35          ) salary_history
 36        FROM job_history jh
 37        WHERE employee#=e.employee#
 38      )
 39      AS job_history_tab
 40    ) job_history
 41  FROM employees e;

View created.
```

Let's retrieve one employee object and look at the query execution plan. No indexes are used in retrieving records from each of the four tables. The cost of the plan is just 1. This is the minimum achievable cost, indicating that there is no join-penalty.

```
SQL> alter session set "_rowsource_execution_statistics"=true;

Session altered.

SQL> SELECT * FROM employees_view WHERE employee# = 1;

         1 IGNATIUS         01-JAN-70
CHILDREN_TAB(CHILDREN_REC('INIGA', '01-JAN-01'), CHILDREN_REC('INIGO', '01-JAN-02'))
JOB_HISTORY_TAB(JOB_HISTORY_REC('01-JAN-91', 'PROGRAMMER', SALARY_HISTORY_TAB(SALARY_HISTORY_REC('01-
FEB-91', 1000), SALARY_HISTORY_
REC('01-MAR-91', 1000))), JOB_HISTORY_REC('01-JAN-92', 'DATABASE ADMIN',
SALARY_HISTORY_TAB(SALARY_HISTORY_REC('01-FEB-92', 2000), S
ALARY_HISTORY_REC('01-MAR-92', 2000))))

SQL> SELECT * FROM TABLE(dbms_xplan.display_cursor(null, null, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------
SQL_ID  aaxmaqz947aa0, child number 0
-------------------------------------
SELECT * FROM employees_view WHERE employee# = 1

Plan hash value: 2117652374

-------------------------------------------------------------------------------------------------------------------
| Id  | Operation         | Name           | Starts | E-Rows |E-Bytes| Cost  | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                |      1 |        |       |     1 |      1 |00:00:00.01 |       1 |
|*  1 |  TABLE ACCESS HASH| CHILDREN       |      1 |      1 |    32 |       |      2 |00:00:00.01 |       1 |
|*  2 |  TABLE ACCESS HASH| SALARY_HISTORY |      2 |      1 |    44 |       |      4 |00:00:00.01 |       3 |
|*  3 |  TABLE ACCESS HASH| JOB_HISTORY    |      1 |      1 |    32 |       |      2 |00:00:00.01 |       1 |
|*  4 |  TABLE ACCESS HASH| EMPLOYEES      |      1 |    845 | 27040 |       |      1 |00:00:00.01 |       1 |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("EMPLOYEE#"=:B1)
   2 - access("EMPLOYEE#"=:B1)
       filter("JOB_DATE"=:B1)
   3 - access("EMPLOYEE#"=:B1)
   4 - access("EMPLOYEE#"=1)

Note
-----
   - cpu costing is off (consider enabling it)
   - dynamic sampling used for this statement (level=2)
```

Next, let's create an "INSTEAD OF" trigger so that we insert into the view *directly*; that is, use a single insert statement instead of multiple insert statements. The trigger will do all the heavy-lifting for us.

```
SQL> CREATE OR REPLACE TRIGGER employees_view_insert
  2  INSTEAD OF INSERT ON employees_view
  3  REFERENCING NEW AS n
  4  FOR EACH ROW
  5  DECLARE
```

```
 6    i NUMBER;
 7  BEGIN
 8    INSERT INTO employees
 9    VALUES
10    (
11      :n.employee#,
12      :n.name,
13      :n.birth_date
14    );
15
16    FOR i IN :n.children.FIRST .. :n.children.LAST
17    LOOP
18      INSERT INTO children
19      VALUES
20      (
21        :n.employee#,
22        :n.children(i).child_name,
23        :n.children(i).birth_date
24      );
25    END LOOP;
26
27    FOR i IN :n.job_history.FIRST .. :n.job_history.LAST
28    LOOP
29      INSERT INTO job_history VALUES
30      (
31        :n.employee#,
32        :n.job_history(i).job_date,
33        :n.job_history(i).title
34      );
35      FOR j IN :n.job_history(i).salary_history.FIRST .. :n.job_history(i).salary_history.LAST
36      LOOP
37        INSERT INTO salary_history
38        VALUES
39        (
40          :n.employee#,
41          :n.job_history(i).job_date,
42          :n.job_history(i).salary_history(j).salary_date,
43          :n.job_history(i).salary_history(j).salary
44         );
45      END LOOP;
46    END LOOP;
47  END;
48  /

  Trigger created.
```

Finally, let's insert an employee object directly into the view and confirm that we can read it back.

```
SQL> INSERT INTO employees_view
  2  VALUES
  3  (
  4    2,
  5    'YGNACIO',
  6    '01-JAN-70',
  7    CHILDREN_TAB
  8    (
  9      CHILDREN_REC('INIGA', '01-JAN-01'),
 10      CHILDREN_REC('INIGO', '01-JAN-02')
 11    ),
 12    JOB_HISTORY_TAB
 13    (
 14      JOB_HISTORY_REC
 15      (
 16        '01-JAN-91',
 17        'PROGRAMMER',
 18        SALARY_HISTORY_TAB
 19        (
 20          SALARY_HISTORY_REC('01-FEB-91', 1000),
 21          SALARY_HISTORY_REC('01-MAR-91', 1000)
 22        )
 23      ),
 24      JOB_HISTORY_REC
```

```
 25      (
 26        '01-JAN-92',
 27        'DATABASE ADMIN',
 28        SALARY_HISTORY_TAB
 29        (
 30          SALARY_HISTORY_REC('01-FEB-92', 2000),
 31          SALARY_HISTORY_REC('01-MAR-92', 2000)
 32        )
 33      )
 34    )
 35  );

1 row created.

SQL> SELECT * FROM employees_view WHERE employee# = 2;

         2 YGNACIO          01-JAN-70
CHILDREN_TAB(CHILDREN_REC('INIGA', '01-JAN-01'), CHILDREN_REC('INIGO', '01-JAN-02'))
JOB_HISTORY_TAB(JOB_HISTORY_REC('01-JAN-91', 'PROGRAMMER', SALARY_HISTORY_TAB(SALARY_HISTORY_REC('01-
FEB-91', 1000), SALARY_HISTORY_
REC('01-MAR-91', 1000))), JOB_HISTORY_REC('01-JAN-92', 'DATABASE ADMIN',
SALARY_HISTORY_TAB(SALARY_HISTORY_REC('01-FEB-92', 2000), S
ALARY_HISTORY_REC('01-MAR-92', 2000))))
```

Schemaless Design

As we said at the outset, NoSQL consists of "disruptive innovations" that are gaining steam and moving upmarket. So far, we have discussed functional segmentation (the pivotal innovation), sharding, asynchronous replication, eventual consistency (resulting from lack of distributed transactions across functional segments and from asynchronous replication), and blobs.

The final innovation of the NoSQL camp is "schemaless design." In database management systems of the NoSQL kind, data is stored in "blobs" and documents the database management system does not police their structure. In mainstream database management systems on the other hand, doctrinal purity requires that the schema be designed *before* data is inserted. Let's do a thought experiment.

Suppose that we don't have a schema and let's suppose that the following facts are known.

- Iggy Fernandez is an employee with EMPLOYEE_ID=1 and SALARY=$1000.
- Mogens Norgaard is a commissioned employee with EMPLOYEE_ID=2, SALARY=€1000, and COMMISSION_PCT=25.
- Morten Egan is a commissioned employee with EMPLOYEE_ID=3, SALARY=€1000, and unknown COMMISSION_PCT.

Could we ask the following questions and expect to receive correct answers?

- **Question**: What is the salary of Iggy Fernandez?
  Expected answer: $1000.
- **Question**: What is the commission percentage of Iggy Fernandez?
  Expected answer: Invalid question.
- **Question**: What is the commission percentage of Mogens Norgaard?
  Expected answer: 25%
- **Question**: What is the commission percentage of Morten Egan?
  Expected answer: Unknown.

If we humans can process the above data and correctly answer the above questions, then surely we can program computers to do so.

The above data could be modeled with the following three relations. It is certainly disruptive to suggest that this be done on the fly by the database management system but not outside the realm of possibility.

```
EMPLOYEES
  EMPLOYEE_ID NOT NULL NUMBER(6)
```

```
    EMPLOYEE_NAME VARCHAR2(128)

  UNCOMMISSIONED_EMPLOYEES
    EMPLOYEE_ID NOT NULL NUMBER(6)
    SALARY NUMBER(8,2)

  COMMISSIONED_EMPLOYEES
    EMPLOYEE_ID NOT NULL NUMBER(6)
    SALARY NUMBER(8,2)
    COMMISSION_PCT NUMBER(2,2)
```

A NoSQL company called Hadapt has already stepped forward with such a feature:

> *"While it is true that SQL requires a schema, it is entirely untrue that the user has to define this schema in advance before query processing.* ***There are many data sets out there, including JSON, XML, and generic key-value data sets that are self-describing — each value is associated with some key that describes what entity attribute this value is associated with*** *[emphasis added]. If these data sets are stored in Hadoop, there is no reason why Hadoop cannot automatically generate a virtual schema against which SQL queries can be issued. And if this is true, users should not be forced to define a schema before using a SQL-on-Hadoop solution — they should be able to effortlessly issue SQL against a schema that was automatically generated for them when data was loaded into Hadoop.*
>
> *Thanks to the hard work of many people at Hadapt from several different groups, including the science team who developed an initial design of the feature, the engineering team who continued to refine the design and integrate it into Hadapt's SQL-on-Hadoop solution, and the customer solutions team who worked with early customers to test and collect feedback on the functionality of this feature, this feature is now available in Hadapt."*
> (http://hadapt.com/blog/2013/10/28/all-sql-on-hadoop-solutions-are-missing-the-point-of-hadoop/)

This is not really new ground. Oracle Database provides the ability to convert XML documents into relational tables (http://docs.oracle.com/cd/E11882_01/appdev.112/e23094/xdb01int.htm#ADXDB0120) though it ought to be possible to view XML data as tables while physically storing it in XML format in order to benefit certain use cases. It should also be possible to redundantly store data in both XML and relational formats in order to benefit other use cases.

In "Extending the Database Relational Model to Capture More Meaning," Dr. Codd explains how a "formatted database" is created from a collection of facts:

> *"Suppose we think of a database initially as a set of formulas in first-order predicate logic. Further, each formula has no free variables and is in as atomic a form as possible (e.g, A & B would be replaced by the component formulas A, B). Now suppose that most of the formulas are simple assertions of the form Pab…z (where P is a predicate and a, b, … , z are constants), and that the number of distinct predicates in the database is few compared with the number of simple assertions. Such a database is usually called formatted, because the major part of it lends itself to rather regular structuring. One obvious way is to factor out the predicate common to a set of simple assertions and then treat the set as an instance of an n-ary relation and the predicate as the name of the relation."*

In other words, a collection of facts can always be organized into relations if necessary.

Oracle NoSQL Database

In May 2011, Oracle Corporation published a scathing indictment of NoSQL, the last words being *"Go for the tried and true path. Don't be risking your data on NoSQL databases."* Just a few months later however, in September of that year, Oracle Corporation released Oracle NoSQL Database. Oracle removed the NoSQL criticism from its website but since information published on the internet is immortal, archived copies can be easily found if you know what you are looking for. In the white paper that accompanied the release of Oracle NoSQL Database, Oracle Corporation claimed that the demands of certain applications could not be met by mainstream database management systems:

> *"The Oracle NoSQL Database, with its "No Single Point of Failure" architecture, is the right solution when data access is "simple" in nature and application demands exceed the volume or latency capability of traditional data management solutions. For example, click-stream data from high volume web sites, high-throughput event processing and social networking communications all represent application domains that produce extraordinary volumes of simple keyed data. Monitoring online retail behavior, accessing customer profiles, pulling up appropriate customer ads and storing and forwarding real-time communication are examples of domains requiring the ultimate in low-latency access. Highly distributed applications such as real-time sensor aggregation and scalable authentication also represent domains well-suited to Oracle NoSQL Database."*

Oracle NoSQL Database has two features that distinguish it from other key-value stores:

- A key is the concatenation of a "major key path" and a "minor key path." All records with the same "major key path" will be colocated on the same storage node.
- Transactional support is provided for modifying multiple records with the same major key path.

Here are some resources to get you started with Oracle NoSQL Database:

- The white paper on Oracle NoSQL Database v2.0; an updated version of the original September 2011 paper.
- Oracle NoSQL Database: Real-Time Big Data Management for the Enterprise by Maqsood Alam, Aalok Muley, Chaitanya Kadaru and Ashok Joshi (Oracle Press, 2013)
- The community edition of Oracle NoSQL Database v2.0 which can be downloaded from http://download.oracle.com/otn-pub/otn_software/nosql-database/kv-ce-2.0.26.zip. The prerequisite is JDK 1.6 or higher which you can download fromhttp://www.oracle.com/technetwork/java/javase/downloads/index.html.
- Installation instructions and a five-minute quickstart for Oracle NoSQL Database are at http://docs.oracle.com/cd/NOSQL/html/quickstart.html. It includes a "Hello World" teaching example illustrating the "put" and "get" function calls which are the basic operations in key-value stores.

```
final String keyString = "Hello";
final String valueString = "Big Data World!";
store.put(Key.createKey(keyString), Value.createValue(valueString.getBytes()));
final ValueVersion valueVersion = store.get(Key.createKey(keyString));
System.out.println(keyString + " " + new String(valueVersion.getValue().getValue()));
```

- Oracle NoSQL Database and Oracle Relational Database – A Perfect Fit, a presentation by Dave Rubin, Director of NoSQL Database development at Oracle Corporation.
- Data Management in Oracle NoSQL Database, a presentation by Anuj Sahni, Principal Product Manager at Oracle Corporation. Also a hands-on database administration workshop.
- The Oracle NoSQL Database resource page on the Oracle Corporation website.

NoSQL Taxonomy

NoSQL databases can be classified into the following categories:

- **Key-value stores**: The archetype is Amazon Dynamo of which DynamoDB is the commercial successor. Key-value stores basically allow applications to "put" and "get" values but each product has differentiators. For example, DynamoDB supports "tables" (namespaces) while Oracle NoSQL Database offers "major" and "minor" key paths.
- **Document stores**: While key-value stores treat values as uninterpreted strings, document stores allow values to be managed using formats such as JSON (JavaScript Object Notation) which are conceptually similar to XML. This allows key-value pairs to be indexed by any component of the value just as XML data can be indexed in mainstream database management systems.
- **Column-family stores**: Column-family stores allow data associated with a single key to be spread over multiple storage nodes. Each storage node only stores a subset of the data associated with the key; hence the name "column-family." A key is therefore composed of a "row key" and a "column key."
- **Graph databases**: Graph databases are non-relational databases that use graph concepts such as nodes and edges to solve certain classes of problems: for example; the shortest route between two towns on a map. The concepts of functional segmentation, sharding, replication, eventual consistency, and schemaless design do not apply to graph databases so I will not discuss graph databases.

NoSQL products are numerous and rapidly evolving. There is a crying need for a continuously updated encyclopedia of NoSQL products but none exists. There is a crying need for an independent benchmarking organization but none exists. My best advice is to do a proof of concept (POC) as well as a PSR (Performance, Scalability, and Reliability) test before committing to using a NoSQL product. Back in the day—in 1985 to be precise—Dr. Codd had words of advice for those who were debating between the new relational products and the established pre-relational products of his day. The advice is as solid today as it was in Dr. Codd's day.

> *"Any buyer confronted with the decision of which DBMS to acquire should weigh three factors heavily.*

> *The first factor is the buyer's performance requirements, often expressed in terms of the number of transactions that must be executed per second. The average complexity of each transaction is also an important consideration. Only if the performance requirements are extremely severe should buyers rule out present relational DBMS products on this basis.* ***Even then buyers should design performance tests of their own, rather than rely on vendor-designed tests or vendor-declared strategies.*** *[emphasis added]*
>
> *The second factor is reduced costs for developing new databases and new application programs …*
>
> *The third factor is protecting future investments in application programs by acquiring a DBMS with a solid theoretical foundation …*
>
> *In every case, a relational DBMS wins on factors two and three. In many cases, it can win on factor one also—in spite of all the myths about performance."*
>
> —An Evaluation Scheme for Database Management Systems that are claimed to be Relational

Big Data in a Nutshell

The topic of Big Data is often brought up in NoSQL discussions so let's give it a nod. In 1998, Sergey Brin and Larry Page invented the PageRank algorithm for ranking web pages (The Anatomy of a Large-Scale Hypertextual Web Search Engine by Brin and Page) and founded Google. The PageRank algorithm required very large matrix-vector multiplications (Mining of Massive Datasets Ch. 5 by Rajaraman and Ullman) so the MapReduce technique was invented to handle such large computations (MapReduce: Simplified Data Processing on Large Clusters). Smart people then realized that the MapReduce technique could be used for other classes of problems and an open-source project called Hadoop was created to popularize the MapReduce technique (The history of Hadoop: From 4 nodes to the future of data). Other smart people realized that MapReduce could handle the operations of relational algebra such as join, anti-join, semi-join, union, difference, and intersection (Mining of Massive Datasets Ch. 2 by Rajaraman and Ullman) and began looking at the possibility of processing large volumes of business data (a.k.a. "Big Data") better and cheaper than mainstream database management systems. Initially programmers had to write Java code for the "mappers" and "reducers" used by MapReduce. However, smart people soon realized that SQL queries could be automatically translated into the necessary Java code and "SQL-on-Hadoop" was born. Big Data thus became about processing large volumes of business data *with SQL* but better and cheaper than mainstream database management systems. However, the smart people have now realized that MapReduce is not the best solution for low-latency queries (Facebook open sources its SQL-on-Hadoop engine, and the web rejoices). Big Data has finally become about processing large volumes of business data with SQL but better and cheaper than mainstream database management systems and *with or without MapReduce.*

That's the fast-moving story of Big Data in a nutshell.

Mistakes of the Relational Camp

Over a lifespan of four and a half decades, the relational camp made a series of strategic mistakes that made NoSQL possible. The mistakes started very early. The biggest mistake is enshrined in the first sentence of the first paper on relational theory by none other than its inventor, Dr. Edgar Codd: *"Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)."* (A Relational Model of Data for Large Shared Data Banks)

How likely is it that application developers will develop scalable high-performance applications if they are shielded from the internal representation of data? SQL was never intended for serious application development. As explained by the creators of SQL in their 1974 paper, there is *"a large class of users who, while they are not computer specialists, would be willing to learn to interact with a computer in a reasonably high-level, non-procedural query language. Examples of such users are* ***accountants, engineers, architects, and urban planners*** *[emphasis added]. It is for this class of users that SEQUEL is intended. For this reason,* ***[SQL] emphasizes simple data structures and operations*** *[emphasis added]."* (http://faculty.cs.tamu.edu/yurttas/PL/DBL/docs/sequel-1974.pdf).

A case in point: If you were the manager of a bookstore, how would you stock the shelves? Would you stand at the door and fling books onto any shelf that had some free space, perhaps recording their locations in a notebook for future reference. Of course not! And would you scatter related books all over the bookstore? Of course not! Then why do we store rows of data in random fashion? The default Oracle table storage structure is the unorganized heap and yet it is chosen 99.9% of the time.

Productivity and ease-of-use were the goals of the relational model, not performance. In Normalized Data Base Structure: A Brief Tutorial (1971), Dr. Codd said *"What is less understandable is the trend toward more and more complexity in the data structures with which application programmers and terminal users directly interact. Surely, in the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance – and that is the convenience of the majority of users. … To make formatted data bases readily accessible to users (especially casual users) who have little or no training in programming we must provide the simplest possible data structures and almost natural language. … What could be a simpler, more universally needed, and more universally understood data structure than a table?"*

Dr. Codd emphasized the productivity benefits of the relational model in his acceptance speech for the 1981 Turing Award: *"It is well known that the growth in demands from end users for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. There are two complementary approaches to attacking this problem (and both approaches are needed): one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs. It is less well known that a single technology, relational database management, provides a practical foundation to both approaches."*

The emphasis on productivity and ease of use at the expense of performance was the biggest mistake of the relational camp.

Mistake #2: Forbidding nested relations

Dr. Codd made a second serious mistake in his 1970 paper by forbidding nested relations in the interests of productivity and ease-of-use. He incorrectly argued (in an unpublished version of that paper) that nested relations would mean that *"The second-order predicate calculus (rather than first-order) is needed because the domains on which relations are defined may themselves have relations as elements."* Not anticipating markup languages like XML, he also argued that *"The simplicity of the array representation which becomes feasible when all relations are cast in normal form is not only an advantage for storage purposes but also for communication of bulk data between systems which use widely different representations of the data."* This caused the detractors of the relational model to observe that *"Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car."* (incorrectly attributed to Esther Dyson, the editor of Release 1.0).

Even though he made a serious mistake by forbidding nested relations, Dr. Codd never said that

- data should be stored in normalized form only;
- each stored table should occupy a separate storage bucket;
- a single data block should only contain data from a single table;
- data should be stored in row-major order; and
- stored tables have only one storage representation each.

Oracle Database has limited support for nested tables but they are not stored inline, they are "collections" not real tables, and you cannot define primary-key or foreign-key constraints on them.

Mistake #3: Favoring relational calculus over relational algebra

At the heart of the relational model are the operations of the "relational algebra": equi-join, theta-join, outer-join, semi-join, anti-join, union, difference, intersection, etc. In the interests of productivity and ease-of-use however, Dr. Codd favored "relational calculus" over relational algebra. In Relational Completeness of Data Base Sublanguages, he proved the equivalence of relational calculus and relational algebra. In the interests of productivity and ease-of-use, Codd then made the decision for you and me: *"Clearly, the majority of users should not have to learn either the relational calculus or algebra in order to interact with data bases. However, requesting data by its properties is far more natural than devising a particular algorithm or sequence of operations for its retrieval. Thus, a calculus-oriented language provides a good target language for a more user-oriented source language."* SQL was therefore based on relational calculus instead of relational algebra and it became the job of the database management system to convert SQL statements into equivalent sequences of relational algebra operations. This is not a trivial problem. It is an *extremely* hard problem. (See Appendix B—The way you write your query matters.)

Mistake #4: Equating the normalized set with the stored set

Equating the normalized set with the stored set ensures that mainstream database management systems will be permanently handicapped in the performance race. For example, a nested relation can be normalized into "first normal form" (no nested

relations). As another example, if B and C are colunm-subsets of A and A = B EQUIJOIN C, then we can perform a "lossless decomposition" of A into B and C. However, the relational model does *not* require that each normalized relation be mapped to a separate bucket of physical storage. Dr. Codd differentiated between the "stored" set, the "named" set, and the "expressible" set but mainstream database management systems conflate the stored set with the normalized set. Mainstream database management systems only allow integrity constraints to be specified on the stored set and only permit DML operations on the stored set. If B and C are column-subsets of A and A = B EQUIJOIN C, then it ought to be possible to store A only while defining primary and foreign key constraints on B and C and allowing DML operations on B and C but this is not permitted by mainstream database management systems.

Mistake #5: Incomplete implementation

Strange as it may sound, the relational model has not yet been fully implemented by mainstream database management systems. You've probably heard of Codd's "twelve rules" published in a 1985 Computerworld article. More than a quarter-century later, mainstream database management systems still do not follow all the twelve rules. DML operations are still not permitted on views (Rule 6 and Rule 7). Arbitrary integrity constraints still cannot be declared and enforced (Rule 10). Integrity constraints still cannot span separate databases (Rule 11). If would be hard to abandon a database management systems that had these capabilities. It is easier to abandon a database management system that does not have these capabilities.

Mistake #6: The marriage of relational and transactional

ACID transactions nothing to do with the relational model per se although relational theory does come in very handy in defining consistency constraints. (See Appendix A—See What's so sacred about relational anyway?) Pre-relational database management systems such as IMS provided ACID guarantees and so did post-relational object-oriented database management systems. We should be able to store non-transactional data outside a transactional database management system while continuing to exploit the entire spectrum of indexing, partitioning, and clustering techniques. (See Appendix C—We don't use databases; we don't use indexes.)

Concluding Remarks

The relational camp put productivity, ease-of-use, and logical elegance front and center. However, the mistakes and misconceptions of the relational camp prevent mainstream database management systems from achieving the performance levels required by modern applications. For example, Dr. Codd forbade nested relations (a.k.a. unnormalized relations) and mainstream database management systems equate the normalized set with the stored set.

The NoSQL camp on the other hand put performance, scalability, and reliability front and center. Understandably the NoSQL camp could not see past the mistakes and misconceptions of the relational camp and lost the opportunity to take the relational model to the next level. Just like the relational camp, the NoSQL camp believes that normalization dictates physical storage choices. Just like the relational camp, the NoSQL camp believes that non-relational APIs are forbidden by the relational model. And the NoSQL camp believes that relational is synonymous with ACID (Atomicity, Consistency, Isolation, Durability).

The NoSQL camp created a number of innovations that are disruptive in the sense used by Harvard Business School professor Clayton Christensen: functional segmentation, sharding, replication, eventual consistency, and schemaless design. Since these innovations are compatible with the relational model, I hope that they will eventually be absorbed by mainstream database management systems.

There are already proofs that performance, scalability, and reliability can be achieved without abandoning the relational model. For example, ScaleBase provides sharding and replication on top of MySQL storage nodes. Another good example to study is VoltDB which claims to be the world's fastest OLTP database (though it has never published an audited TPC-C benchmark). A counter-example to Amazon is eBay which arguably has equal scale and equally high performance, scalability, and reliability requirements. eBay uses performance segmentation, sharding, replication, and eventual consistency but continues to use Oracle (and SQL) to manage the local database. I asked Randy Shoup, one of the architects of the eBay e-commerce platform, why eBay did not abandon Oracle Database and he answered in one word: "comfort." Here are links to some of his presentations and articles on the eBay architecture:

- eBay's Scaling Odyssey: Growing and Evolving a Large eCommerce Site (Slide deck)
- The eBay Architecture: Striking a balance between site stability, feature velocity, performance, and cost (Slide deck)
- Randy Shoup Discusses the eBay Architecture (video and transcript)

- Randy Shoup on eBay's Architectural Principles (video and transcript)
- Scalability Best Practices: Lessons from eBay (blog post)

Finally, I should point out that are very good reasons to criticize current NoSQL products; for example, lack of standards, primitive feature sets, primitive security, and primitive management tools, unproven claims, and traps for the unwary. MongoDB uses a database-wide lock for reads and writes …

Appendix A—What's so sacred about relational anyway?

Dr. Codd personally believed that the chief advantage of the relational model was its simplicity and consequent appeal to users (especially casual users) who have little or no training in programming. He singles out this advantage in the opening sentence of his very first paper on relational theory, *A Relational Model of Data for Large Shared Data Banks,* faithfully reproduced in the 100th issue of the *NoCOUG Journal* (down to the misspelling of the city name Phoenix in the References section): "*Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation).*" He makes the point very forcefully in a subsequent paper, *Normalized Data Base Structure: A Brief Tutorial*: *"In the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance—and that is the convenience of the majority of users. … To make formatted data bases readily accessible to users (especially casual users) who have little or no training in programming we must provide the simplest possible data structures and almost natural language. … What could be a simpler, more universally needed, and more universally understood data structure than a table? Why not permit such users to view all the data in a data base in a tabular way?"*

But does the appeal to users (especially casual users) who have little or no training in programming make relational sacred to computer professionals? Should computer professionals like you and me be protected from having to know how the data is organized in the machine? Will we develop high-performance applications if we are ignorant about those little details? If your answers are in the negative, then read on.

Computational Elegance

Dividing 3704 by 14 is computationally more elegant than dividing MMMDCCIV by XIV (Roman notation), wouldn't you agree? (Here's how the Romans did division). The computational elegance of the relational model is unquestionable. The co-inventor of the SQL Language, Donald Chamberlin, reminisces: *"Codd gave a seminar and a lot of us went to listen to him. This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I'd been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." He just whacked them out and you could sort of read them, and they weren't complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that."* (The 1995 SQL Reunion: People, Projects, and Politics)

But is computational elegance the holy grail of computer professionals? Is it the be-all and end-all of application software development? If your answers are in the negative, then read on.

Derivability, Redundancy, and Consistency of Relations

The true importance of relational theory is highlighted by the title of the original (and shorter) version of Codd's first paper. That version predated the published version by a year, and the title was *"Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks."* The title of this unpublished version emphasizes that the real importance of relational theory is that it provides a rigorous method of asserting arbitrarily complex consistency constraints that must be satisfied by the data within the database. For example, the following assertion written in Structured Query Language (SQL) states that the company must have at least 50 employees:

```
CREATE ASSERTION employees_a1 AS CHECK (
  (SELECT COUNT(*) FROM employees) >= 50
)
```

In practice, the above consistency check would have to be implemented using a database trigger, since relational vendors do not support the CREATE ASSERTION feature of SQL-92.

As another example, the following "referential integrity constraint" links the Employees table to the Departments table:

```
CREATE ASSERTION emp_dept_fk AS CHECK (
  NOT EXISTS (
```

```
      SELECT * FROM employees e WHERE NOT EXISTS (
        SELECT * FROM departments d WHERE d.department_id = e.department_id
      )
    )
  )
```

In practice, referential integrity constraints are implemented using simplified syntax that obfuscates the theoretical underpinnings. The database administrator or application developer need simply say:

```
ALTER TABLE employees ADD CONSTRAINT emp_dept_fk
FOREIGN KEY ("DEPARTMENT_ID") REFERENCES departments;
```

Strange as it may sound, Oracle Database did not enforce referential integrity constraints until Version 7 was released in the 1990s (by which time Oracle Corporation was already the world's largest database company). From the January 1989 issue of *Software Magazine*:

> *"About six or seven years ago when I worked for a vendor that made a Codasyl DBMS called Seed, I spoke at a conference. Also speaking was Larry Rowe, one of the founders of Relational Technology, Inc. and one of the developers of the relational DBMS Ingres. We were about to be clobbered by these new relational systems. He suggested to me that the best way to compete against the relational systems was to point out that they did not support referential integrity. Well, back then, virtually no one understood the problem enough to make it an issue. Today, Codasyl DBMSs are an endangered species, and referential integrity is a hot topic used by the relational DBMSs to compete among themselves."*

To summarize, the relational model is sacred because it gives application software developers the ability to assert and enforce consistency of data in databases.

Appendix B—The way you write your query matters

In 1988, a SQL researcher named Fabian Pascal wrote an article for Database Programming and Design in which he quoted Chris Date as saying:

> *"SQL is an extremely redundant language. By this I mean that all but the most trivial of problems can be expressed in SQL in a variety of different ways. Of course, the differences would not be important if all formulations worked equally well but that is unlikely. As a result, users are forced to spend time and effort trying to find the "best" formulation (that is, the version that performs best)—which is exactly one of the things the relational model was trying to avoid in the first place."*

Pascal then went on to test seven equivalent queries with five different database engines. Only one out of the five database engines came anywhere near to acing the test; it used the same execution plan for six of the queries but did not support the seventh formulation. The other engines used a range of query plans with different execution times. Pascal then predicted that:

> *"Eventually, all SQL DBMSs, for competitive reasons, will have to equalize the performance of redundant SQL expressions and to document their execution plans. Forcing users to maximize performance through query formulation is not only unproductive, but simply a lost cause, especially if there is no guidance from the system. The more users understand the relational model and its productivity intentions, the more they will demand equalized performance and documented execution plans from vendors, instead of doggedly attempting to undertake unnecessary and futile burdens."*

Let's find out if Pascal's twenty-five year old prediction of equalized performance came true. First, let's create tables similar to those that Pascal used. The tests require a table called Personnel containing employee details and a table called Payroll containing salary payments. The employee ID is the primary key of each table (meaning that they are also indexed using the employee ID) and the payroll table is indexed using salary. The problem is to find out which one of the 9900 employees received a payment of $199,170. I conducted my tests using Oracle Database 11*g* Release 2.

```
CREATE TABLE personnel
(
  empid CHAR(9),
  lname CHAR(15),
  fname CHAR(12),
  address CHAR(20),
  city CHAR(20),
  state CHAR(2),
  ZIP CHAR(5)
);

CREATE TABLE payroll
```

```
(
  empid CHAR(9),
  bonus INTEGER,
  salary INTEGER
);

INSERT INTO personnel
SELECT
  TO_CHAR(LEVEL, '09999') AS empid,
  DBMS_RANDOM.STRING('U', 15) AS lname,
  DBMS_RANDOM.STRING('U', 12) AS fname,
  '500 ORACLE PARKWAY' AS address,
  'REDWOOD SHORES' AS city,
  'CA' AS state,
  '94065' AS zip
FROM
  dual
CONNECT BY LEVEL <= 9900;

INSERT INTO payroll(empid, bonus, salary)
SELECT
  per.empid,
  0 as bonus,
  99170 + ROUND(DBMS_RANDOM.VALUE * 100000, -2) AS salary
FROM
  personnel per;

CREATE UNIQUE INDEX personnel_u1 ON personnel(empid);
CREATE UNIQUE INDEX payroll_u1 ON payroll(empid);
CREATE INDEX payroll_i1 ON payroll(salary);

EXEC DBMS_STATS.GATHER_TABLE_STATS(ownname=>'HR', tabname=>'PERSONNEL');
EXEC DBMS_STATS.GATHER_TABLE_STATS(ownname=>'HR', tabname=>'PAYROLL');
```

Observe that I gathered optimizer statistics for both tables although they are not strictly necessary for the tests that follow; a query optimizer can generate query plans even in the absence of statistics.

Relational algebra method

The first formulation uses the relational algebra method. The query optimizer drives from the payroll table to the personnel table, a good move.

```
SELECT DISTINCT per.empid, per.lname
FROM personnel per JOIN payroll pay ON (per.empid = pay.empid)
WHERE pay.salary = 199170;

EMPID     LNAME
--------- ---------------
 01836    KULGDTAFIIYUDIE
 04535    ZZNDFPAGHWQAVSV
 07751    EFBDSEXSBJUQJIF
 06679    CCAZNDOPKSKEQRS

SQL_ID  cx451qsx2qfcv, child number 0
-------------------------------------
SELECT DISTINCT per.empid, per.lname FROM personnel per JOIN payroll
pay ON (per.empid = pay.empid) WHERE pay.salary = 199170

Plan hash value: 3901981856

-----------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |      1 |        |       |    21 (100)|          |      4 |00:00:00.01 |      16 |
|   1 |  HASH UNIQUE                  |              |      1 |     10 |   410 |    21   (5)| 00:00:01 |      4 |00:00:00.01 |      16 |
|   2 |   NESTED LOOPS                |              |      1 |        |       |            |          |      4 |00:00:00.01 |      16 |
|   3 |    NESTED LOOPS               |              |      1 |     10 |   410 |    20   (0)| 00:00:01 |      4 |00:00:00.01 |      12 |
|   4 |     TABLE ACCESS BY INDEX ROWID| PAYROLL      |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|*  5 |      INDEX RANGE SCAN         | PAYROLL_I1   |      1 |     10 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       2 |
|*  6 |     INDEX UNIQUE SCAN         | PERSONNEL_U1 |      4 |      1 |       |     0   (0)|          |      4 |00:00:00.01 |       6 |
|   7 |    TABLE ACCESS BY INDEX ROWID| PERSONNEL    |      4 |      1 |    26 |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("PAY"."SALARY"=199170)
   6 - access("PER"."EMPID"="PAY"."EMPID")
```

Uncorrelated subquery

The second formulation uses an uncorrelated subquery. Even though the form of the query suggests that the optimizer will drive from the personnel table to the payroll table, the optimizer instead chooses to drive the other way. It is also smart enough to realize that duplicate rows will not be introduced and it therefore dispenses with the deduplication operation seen in the previous execution plan.

```
SELECT per.empid, per.lname
FROM personnel per
WHERE per.empid IN (SELECT pay.empid
FROM payroll pay
WHERE pay.salary = 199170);

EMPID      LNAME
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 06679     CCAZNDOPKSKEQRS
 07751     EFBDSEXSBJUQJIF

SQL_ID  avhtrqsvaay7j, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE per.empid IN
(SELECT pay.empid FROM payroll pay WHERE pay.salary = 199170)

Plan hash value: 3342999746

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |      1 |        |       |    20 (100)|          |      4 |00:00:00.01 |      17 |
|   1 |  NESTED LOOPS                 |               |      1 |        |       |            |          |      4 |00:00:00.01 |      17 |
|   2 |   NESTED LOOPS                |               |      1 |     10 |   410 |    20   (0)| 00:00:01 |      4 |00:00:00.01 |      13 |
|   3 |    TABLE ACCESS BY INDEX ROWID| PAYROLL       |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       7 |
|*  4 |     INDEX RANGE SCAN          | PAYROLL_I1    |      1 |     10 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       3 |
|*  5 |    INDEX UNIQUE SCAN          | PERSONNEL_U1  |      4 |      1 |       |     0   (0)|          |      4 |00:00:00.01 |       6 |
|   6 |   TABLE ACCESS BY INDEX ROWID | PERSONNEL     |      4 |      1 |    26 |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("PAY"."SALARY"=199170)
```

Correlated subquery

The third formulation uses the relational calculus method. This time too, the query optimizer drives from the payroll table to the personnel table but this time it thinks that a deduplication operation is necessary.

```
SELECT per.empid, per.lname
FROM personnel per
WHERE EXISTS (SELECT *
  FROM payroll pay
  WHERE per.empid = pay.empid
  AND pay.salary = 199170);

EMPID      LNAME
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 06679     CCAZNDOPKSKEQRS
 07751     EFBDSEXSBJUQJIF

SQL_ID  gdazhxm5xdu44, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE EXISTS (SELECT *
FROM payroll pay   WHERE per.empid = pay.empid   AND pay.salary =
199170)

Plan hash value: 864898783

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |      1 |        |       |    16 (100)|          |      4 |00:00:00.01 |      16 |
|   1 |  NESTED LOOPS                 |               |      1 |        |       |            |          |      4 |00:00:00.01 |      16 |
|   2 |   NESTED LOOPS                |               |      1 |     10 |   410 |    16   (7)| 00:00:01 |      4 |00:00:00.01 |      12 |
|   3 |    SORT UNIQUE                |               |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|   4 |     TABLE ACCESS BY INDEX ROWID| PAYROLL       |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|*  5 |      INDEX RANGE SCAN         | PAYROLL_I1    |      1 |     10 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       2 |
|*  6 |    INDEX UNIQUE SCAN          | PERSONNEL_U1  |      4 |      1 |       |     0   (0)|          |      4 |00:00:00.01 |       6 |
|   7 |   TABLE ACCESS BY INDEX ROWID | PERSONNEL     |      4 |      1 |    26 |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       4 |
```

```
-----------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("PAY"."SALARY"=199170)
   6 - access("PER"."EMPID"="PAY"."EMPID")
```

Scalar subquery in the WHERE clause

The fourth formulation uses a scalar subquery in the WHERE clause. Oracle performs the scalar subquery once for each employee in the payroll table which is terribly inefficient.

```
SELECT per.empid, per.lname
FROM personnel per
WHERE (SELECT pay.salary FROM payroll pay WHERE pay.empid = per.empid) = 199170;

EMPID      LNAME
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 06679     CCAZNDOPKSKEQRS
 07751     EFBDSEXSBJUQJIF

SQL_ID  ddgmw1whng5ah, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE (SELECT pay.salary
FROM payroll pay WHERE pay.empid = per.empid) = 199170

Plan hash value: 3607962630

-----------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |      1 |        |       | 11133 (100)|          |      4 |00:00:01.74 |   10551 |
|*  1 |  FILTER                      |            |      1 |        |       |            |          |      4 |00:00:01.74 |   10551 |
|   2 |   TABLE ACCESS FULL          | PERSONNEL  |      1 |   9900 |   251K|    68   (0)| 00:00:01 |   9900 |00:00:00.13 |     204 |
|   3 |   TABLE ACCESS BY INDEX ROWID| PAYROLL    |   9900 |      1 |    15 |     2   (0)| 00:00:01 |   9900 |00:00:00.98 |   10347 |
|*  4 |    INDEX UNIQUE SCAN         | PAYROLL_U1 |   9900 |      1 |       |     1   (0)| 00:00:01 |   9900 |00:00:00.34 |     447 |
-----------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(=199170)
   4 - access("PAY"."EMPID"=:B1)
```

Scalar subquery in the SELECT clause

The fifth formulation uses a scalar subquery in the SELECT clause. The query optimizer executed the scalar subquery once for each salary of 199170. Step 1 is executed for each row returned by step 3. The number of buffers reported for line 0 should therefore be 26 not 16.

```
SELECT DISTINCT pay.empid, (SELECT lname FROM personnel per WHERE per.empid = pay.empid)
FROM payroll pay
WHERE pay.salary = 199170;

EMPID      (SELECTLNAMEFRO
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 07751     EFBDSEXSBJUQJIF
 06679     CCAZNDOPKSKEQRS

SQL_ID  6y4kznqkvq635, child number 0
-------------------------------------
SELECT DISTINCT pay.empid, (SELECT lname FROM personnel per WHERE
per.empid = pay.empid) FROM payroll pay WHERE pay.salary = 199170

Plan hash value: 750911849

-----------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |      1 |        |       |    11 (100)|          |      4 |00:00:00.01 |      16 |
|   1 |  TABLE ACCESS BY INDEX ROWID | PERSONNEL    |      4 |      1 |    26 |     2   (0)| 00:00:01 |      4 |00:00:00.01 |      10 |
|*  2 |   INDEX UNIQUE SCAN          | PERSONNEL_U1 |      4 |      1 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|   3 |  HASH UNIQUE                 |              |      1 |     10 |   150 |    11  (10)| 00:00:01 |      4 |00:00:00.01 |      16 |
|   4 |   TABLE ACCESS BY INDEX ROWID| PAYROLL      |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|*  5 |    INDEX RANGE SCAN          | PAYROLL_I1   |      1 |     10 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       2 |
-----------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - access("PER"."EMPID"=:B1)
   5 - access("PAY"."SALARY"=199170)
```

Aggregate function to check existence

The sixth formulation uses the COUNT aggregate function to check existence. I obtained surprising results; the execution plan was terribly inefficient when the constant was placed on the right hand side of the condition but was superbly efficient when the constant was placed on the left hand side. The efficient result testifies to the power of the optimizer while the inefficient result testifies to its fallibility.

```
SELECT per.empid, per.lname
FROM personnel per
WHERE (SELECT count(*) FROM payroll pay WHERE pay.empid = per.empid AND pay.salary = 199170) > 0;

EMPID      LNAME
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 06679     CCAZNDOPKSKEQRS
 07751     EFBDSEXSBJUQJIF

SQL_ID  9df084bq799p1, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE (SELECT count(*)
FROM payroll pay WHERE pay.empid = per.empid AND pay.salary = 199170) >
0

Plan hash value: 3561519015

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |      1 |        |       |  9970 (100)|          |      4 |00:00:01.71 |   10555 |
|*  1 |  FILTER                       |            |      1 |        |       |            |          |      4 |00:00:01.71 |   10555 |
|   2 |   TABLE ACCESS FULL           | PERSONNEL  |      1 |   9900 |   251K|    68   (0)| 00:00:01 |   9900 |00:00:00.09 |     204 |
|   3 |   SORT AGGREGATE              |            |   9900 |      1 |    15 |            |          |   9900 |00:00:01.20 |   10351 |
|*  4 |    TABLE ACCESS BY INDEX ROWID| PAYROLL    |   9900 |      1 |    15 |     2   (0)| 00:00:01 |      4 |00:00:00.73 |   10351 |
|*  5 |     INDEX UNIQUE SCAN         | PAYROLL_U1 |   9900 |      1 |       |     1   (0)| 00:00:01 |   9900 |00:00:00.27 |     451 |
-----------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(>0)
   4 - filter("PAY"."SALARY"=199170)
   5 - access("PAY"."EMPID"=:B1)

SELECT per.empid, per.lname
FROM personnel per
WHERE 0 < (SELECT count(*) FROM payroll pay WHERE pay.empid = per.empid AND pay.salary = 199170);

EMPID      LNAME
---------- ---------------
 01836     KULGDTAFIIYUDIE
 04535     ZZNDFPAGHWQAVSV
 06679     CCAZNDOPKSKEQRS
 07751     EFBDSEXSBJUQJIF

SQL_ID  8bk6d3udbcbp4, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE 0 < (SELECT
count(*) FROM payroll pay WHERE pay.empid = per.empid AND pay.salary =
199170)

Plan hash value: 864898783

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |      1 |        |       |    16 (100)|          |      4 |00:00:00.01 |      16 |
|   1 |  NESTED LOOPS                 |              |      1 |        |       |            |          |      4 |00:00:00.01 |      16 |
|   2 |   NESTED LOOPS                |              |      1 |     10 |   410 |    16   (7)| 00:00:01 |      4 |00:00:00.01 |      12 |
|   3 |    SORT UNIQUE                |              |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|   4 |     TABLE ACCESS BY INDEX ROWID| PAYROLL      |      1 |     10 |   150 |    10   (0)| 00:00:01 |      4 |00:00:00.01 |       6 |
|*  5 |      INDEX RANGE SCAN         | PAYROLL_I1   |      1 |     10 |       |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       2 |
|*  6 |    INDEX UNIQUE SCAN          | PERSONNEL_U1 |      4 |      1 |       |     0   (0)|          |      4 |00:00:00.01 |       6 |
|   7 |   TABLE ACCESS BY INDEX ROWID | PERSONNEL    |      4 |      1 |    26 |     1   (0)| 00:00:01 |      4 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("PAY"."SALARY"=199170)
   6 - access("PAY"."EMPID"="PER"."EMPID")
```

Correlated subquery (double negative)

You may have thought to ask: how does one know *for sure* that all the above queries are equivalent. The answer is that one would have to convert them to a "canonical" form. All semantically equivalent queries would be converted into the same canonical form. It's a mechanical chore that could be automated but unfortunately such an automation tool does not exist. Here are two more formulations with different query plans. Are they semantically equivalent to the previous queries?

```
SELECT per.empid, per.lname
FROM personnel per
WHERE NOT EXISTS (SELECT *
 FROM payroll pay
 WHERE pay.empid = per.empid
 AND pay.salary != 199170);

EMPID     LNAME
--------- ---------------
 01836    KULGDTAFIIYUDIE
 04535    ZZNDFPAGHWQAVSV
 06679    CCAZNDOPKSKEQRS
 07751    EFBDSEXSBJUQJIF

SQL_ID  ayvvv10ah456y, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE NOT EXISTS (SELECT
*  FROM payroll pay  WHERE pay.empid = per.empid  AND pay.salary !=
199170)

Plan hash value: 103534934

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |      1 |        |       |    80 (100)|          |      4 |00:00:00.71 |     241 |
|*  1 |  HASH JOIN RIGHT ANTI|           |      1 |     99 |  4059 |    80   (2)| 00:00:01 |      4 |00:00:00.71 |     241 |
|*  2 |   TABLE ACCESS FULL  | PAYROLL   |      1 |   9890 |   144K|    11   (0)| 00:00:01 |   9896 |00:00:00.06 |      37 |
|   3 |   TABLE ACCESS FULL  | PERSONNEL |      1 |   9900 |   251K|    68   (0)| 00:00:01 |   9900 |00:00:00.06 |     204 |
-----------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("PAY"."EMPID"="PER"."EMPID")
   2 - filter("PAY"."SALARY"<>199170)
```

Uncorrelated subquery (double negative)

```
SELECT per.empid, per.lname
FROM personnel per
WHERE per.empid NOT IN (SELECT pay.empid
FROM payroll pay
WHERE pay.salary != 199170);

EMPID     LNAME
--------- ---------------
 01836    KULGDTAFIIYUDIE
 04535    ZZNDFPAGHWQAVSV
 06679    CCAZNDOPKSKEQRS
 07751    EFBDSEXSBJUQJIF

SQL_ID  67azvy1nw1am1, child number 0
-------------------------------------
SELECT per.empid, per.lname FROM personnel per WHERE per.empid NOT IN
(SELECT pay.empid FROM payroll pay WHERE pay.salary != 199170)

Plan hash value: 2202369223

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name      | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |      1 |        |       |    80 (100)|          |      4 |00:00:00.53 |     241 |
|*  1 |  HASH JOIN RIGHT ANTI NA|           |      1 |     99 |  4059 |    80   (2)| 00:00:01 |      4 |00:00:00.53 |     241 |
|*  2 |   TABLE ACCESS FULL    | PAYROLL   |      1 |   9890 |   144K|    11   (0)| 00:00:01 |   9896 |00:00:00.07 |      37 |
|   3 |   TABLE ACCESS FULL    | PERSONNEL |      1 |   9900 |   251K|    68   (0)| 00:00:01 |   9900 |00:00:00.06 |     204 |
-----------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("PER"."EMPID"="PAY"."EMPID")
   2 - filter("PAY"."SALARY"<>199170)
```

Here are the elapsed time (microseconds) and consistent gets for each of the queries, sorted by elapsed time.

```
METHOD                                SQL_ID        PLAN_HASH_VALUE LAST_ELAPSED_TIME LAST_CR_BUFFER_GETS
------------------------------------- ------------- --------------- ----------------- -------------------
Uncorrelated subquery                 avhtrqsvaay7j      3342999746               129                  17
Aggregate function to check existence 8bk6d3udbcbp4       864898783               135                  16
Correlated subquery                   gdazhxm5xdu44       864898783               405                  16
Relational algebra method             cx451qsx2qfcv      3901981856               426                  16
Scalar subquery in the SELECT clause  6y4kznqkvq635       750911849               701                  16
Uncorrelated subquery (double negative) 67azvy1nw1am1    2202369223              7702                 241
Correlated subquery (double negative) ayvvv10ah456y       103534934             14499                  241
Scalar subquery in the WHERE clause   ddgmw1whng5ah      3607962630            195999                10549
Aggregate function to check existence 9df084bq799p1      3561519015            310690                10554
```

Appendix C—We don't use databases; we don't use indexes

Mogens Norgaard is the co-founder of the OakTable Network, which bills itself as "a network for the Oracle *scientist*, who believes in better ways of administering and developing Oracle-based systems." Whenever salespeople phone him, he claims that he puts them off by saying that he just doesn't use the products that they are calling about.

When the office furniture company phones, he says *"We don't use office furniture."* When the newspaper company phones, he says *"We don't read newspapers."* When the girl scouts phone, he probably says *"We don't eat cookies."*

Once he got a phone call from the *phone* company.

You can only imagine how *that* conversation went. Read the whole story at http://wedonotuse.com/stories-and-answers.aspx.

I wonder what Mogens would say if a database vendor phoned. I can imagine him saying *"We don't use databases. We don't use indexes. We store all our data in compressed text files. Each compressed text file contains one year of data for one location. There is a separate subdirectory for each year. We have a terabyte of data going back to 1901 so we currently have 113 subdirectories. The performance is just fine, thank you."*

On second thoughts, that's just too far-fetched.

You see, back in the early days of the relational era, the creator of relational theory, Dr. Edward Codd married relational theory with transactional database management systems (a.k.a. ACID DBMS) and the Relational Database Management System (RDBMS) was born. He authored two influential ComputerWorld articles—*"Is your DBMS really relational?"* (October 14, 1985) and *"Does your DBMS run by the rules?"* (October 21, 1985)—that set the direction of the relational movement for the next quarter century. Today, the full declarative power of "data base sublanguages" (the term coined by Dr. Codd) such as Structured Query Language (SQL) is only available within the confines of a transactional database management system.

But it shoudn't have to be that way.

Consider the running example of "big data" used in *Hadoop: The Definitive Guide*. The National Climatic Data Center publishes hourly climatic data such as temperature and pressure from more than 10,000 recording stations all over the world. Data from 1901 onwards is available in text files. Each line of text contains the station code, the timestamp, and a number of climatic readings. The format is documented at ftp://ftp.ncdc.noaa.gov/pub/data/noaa/ish-format-document.pdf. The files are organized into subdirectories, one subdirectory for each year. Each subdirectory contains one file from each recording station that was in operation during that year. The individual files are compressed using gzip. All the files can be downloaded from ftp://ftp.ncdc.noaa.gov/pub/data/noaa/.

You might have already guessed where I am going with this.

Conceptually the above terabyte-sized data set is a single table. But it should not be necessary to uncompress and load this huge quantity of structured non-transactional data into a *transactional* database management system in order to query it. The choice of physical representation conserves storage space and is a technical detail that is irrelevant to the logical presentation of the data set as a single table; it is a technical detail that users don't care about. As Dr. Codd said in the opening sentence of his 1970 paper *A Relational Model of Data For Large Shared Data Banks* (faithfully reproduced in the 100th issue of the *NoCOUG Journal*), *"future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)."*

Why shouldn't we be able to query the above data set using good old SQL?

Well you *can* do just that with the Oracle query engine and you *don't* have to load it into an Oracle database first. You can even take advantage of partitioning and parallelism. You can also write queries that mix and match data from the database and the filesystem.

The following demonstrations were performed using a pre-Built developer VM for Oracle VM VirtualBox. The version of Oracle Database is 12.1.0.1.

```
SQL*Plus: Release 12.1.0.1.0 Production on Fri Aug 16 1

0:45:04 2013

Copyright (c) 1982, 2013, Oracle.  All rights reserved.
```

In the demonstrations, we only consider the years from 1901 to 1904. Here is the directory structure.

```
/home/oracle/app/oracle/admin/orcl/dpdump/noaa
/home/oracle/app/oracle/admin/orcl/dpdump/noaa/1904
/home/oracle/app/oracle/admin/orcl/dpdump/noaa/1902
/home/oracle/app/oracle/admin/orcl/dpdump/noaa/1903
/home/oracle/app/oracle/admin/orcl/dpdump/noaa/1901
```

We first need to create "directories" and define an "external table." The definition of this external table specifies a preprocessing script which is the secret sauce that makes it possible for the query engine to traverse the subdirectories and uncompress the data.

```
connect / as sysdba

alter session set container=pdborcl;

create or replace directory share_dir
  as '/u01/app/oracle/admin/orcl/share';

create or replace directory noaa_dir
  as '/u01/app/oracle/admin/orcl/share/noaa';

create or replace directory noaa_1901_dir
  as '/u01/app/oracle/admin/orcl/share/noaa_1901';

create or replace directory noaa_1902_dir
  as '/u01/app/oracle/admin/orcl/share/noaa_1902';

create or replace directory noaa_1903_dir
  as '/u01/app/oracle/admin/orcl/share/noaa_1903';

create or replace directory noaa_1904_dir
  as '/u01/app/oracle/admin/orcl/share/noaa_1904';

grant all on directory share_dir to public;
grant all on directory noaa_dir to public;
grant all on directory noaa_1901_dir to public;
grant all on directory noaa_1902_dir to public;
grant all on directory noaa_1903_dir to public;
grant all on directory noaa_1904_dir to public;

connect iggy/iggy@pdborcl

drop table temperatures;
create table temperatures
(
  station_code char(6),
  datetime char(12),
  temperature char(5)
)
organization external
(
  type oracle_loader
  default directory share_dir
  access parameters
  (
    records delimited by newline
    preprocessor share_dir:'uncompress.sh'
    fields
    (
      station_code position(1:6) char(4),
      datetime position(7:18) char(12),
      temperature position(19:23) char(5)
```

```
    )
  )
  location ('noaa')
);
```

Here's the tiny preprocessing script that makes it possible for Oracle to traverse the subdirectories and uncompress the data. It recursively traverses the file system beginning with the location specified by the query engine; that is, the location specified in the table definition. It uncompresses all zipped files it finds and sends the output to the "cut" utility which cuts out only those column positions that we care about and writes what's left to standard output, not to the filesystem.

```
#!/bin/sh
/usr/bin/find $1 -name "*.gz" -exec /bin/zcat {} \; | /usr/bin/cut -c5-10,16-27,88-92
```

All the capabilities of SQL—including analytic functions and pivoting—can now be exploited as shown in the following example. For each month in the year 1901, we list the top three recording stations in terms of average monthly temperature.

```
set pagesize 66
select /*+ gather_plan_statistics */ * from
(
  select
    month,
    station_code,
    dense_rank() over (partition by month order by average) as rank
  from
  (
    select
      substr(datetime,1,4)||'/'||substr(datetime,5,2) as month,
      station_code,
      avg(temperature) as average
    from temperatures
    where datetime >= '1901' and datetime < '1902'
    group by
      substr(datetime,1,4)||'/'||substr(datetime,5,2),
      station_code
  )
)
pivot(max(station_code) for rank in (1, 2, 3))
order by month;

MONTH                  1      2      3
---------------------- ------ ------ ------
1901/01                2270   0296   0297
1901/02                2270   0290   0296
1901/03                2270   0290   0296
1901/04                0290   0295   0298
1901/05                0290   0295   0298
1901/06                0290   0298   0295
1901/07                0290   0295   2270
1901/08                2270   0290   0296
1901/09                0290   2270   0296
1901/10                2270   0296   0290
1901/11                2270   0296   0297
1901/12                2270   0296   0290
```

We can also use "partition views" and take advantage of "partition pruning." For those who don't remember, partition views are a really old feature that predates "real" partitioning in Oracle 8.0 and above. Partition views continue to work just fine today, even in Oracle Database 12*c*.

Let's create a separate table definition for each year and then use a partition view to tie the tables together.

```
create table temperatures_1901
(
  station_code char(6),
  datetime char(12),
  temperature char(5)
)
organization external
(
  type oracle_loader
  default directory noaa_dir
  access parameters
  (
    records delimited by newline
```

```
    preprocessor share_dir:'uncompress.sh'
    fields
    (
      station_code position(1:6) char(4),
      datetime position(7:18) char(12),
      temperature position(19:23) char(5)
    )
  )
  location ('1901')
);

-- the remaining table definitions are not shown for brevity

create or replace view temperatures_v as
select * from temperatures_1901
where datetime >= '190101010000' and datetime < '190201010000'
  union all
select * from temperatures_1902
where datetime >= '190201010000' and datetime < '190301010000'
  union all
select * from temperatures_1903
where datetime >= '190301010000' and datetime < '190401010000'
  union all
select * from temperatures_1904
where datetime >= '190401010000' and datetime < '190501010000';
```

When we specify only a portion of the temperatures_v view, the query plan confirms that the unneeded branches of the view are filtered out by the query optimizer.

```
select /*+ gather_plan_statistics */ * from
(
  select
    month,
    station_code,
    dense_rank() over (partition by month order by average) as rank
  from
  (
    select
      substr(datetime,1,4)||'/'||substr(datetime,5,2) as month,
      station_code,
      avg(temperature) as average
    from temperatures_v
    where datetime >= '190101010000' and datetime < '190201010000'
    group by
      substr(datetime,1,4)||'/'||substr(datetime,5,2),
      station_code
  )
)
pivot(max(station_code) for rank in (1, 2, 3))
order by month;

Plan hash value: 2790062116

-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name               | Starts | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                    |      1 |     12 |00:00:00.11 |      64 |
|   1 |  SORT GROUP BY PIVOT               |                    |      1 |     12 |00:00:00.11 |      64 |
|   2 |   VIEW                             |                    |      1 |     72 |00:00:00.11 |      64 |
|   3 |    WINDOW SORT                     |                    |      1 |     72 |00:00:00.11 |      64 |
|   4 |     HASH GROUP BY                  |                    |      1 |     72 |00:00:00.11 |      64 |
|   5 |      VIEW                          | TEMPERATURES_V     |      1 |   6565 |00:00:01.43 |      64 |
|   6 |       UNION-ALL                    |                    |      1 |   6565 |00:00:00.48 |      64 |
|*  7 |        EXTERNAL TABLE ACCESS FULL  | TEMPERATURES_1901  |      1 |   6565 |00:00:00.34 |      64 |
|*  8 |        FILTER                      |                    |      1 |      0 |00:00:00.01 |       0 |
|*  9 |         EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1902  |      0 |      0 |00:00:00.01 |       0 |
|* 10 |        FILTER                      |                    |      1 |      0 |00:00:00.01 |       0 |
|* 11 |         EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1903  |      0 |      0 |00:00:00.01 |       0 |
|* 12 |        FILTER                      |                    |      1 |      0 |00:00:00.01 |       0 |
|* 13 |         EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1904  |      0 |      0 |00:00:00.01 |       0 |
-----------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   7 - filter(("DATETIME">='190101010000' AND "DATETIME"<'190201010000'))
   8 - filter(NULL IS NOT NULL)
   9 - filter(("DATETIME">='190201010000' AND "DATETIME"<'190201010000'))
  10 - filter(NULL IS NOT NULL)
  11 - filter(("DATETIME">='190301010000' AND "DATETIME"<'190201010000'))
  12 - filter(NULL IS NOT NULL)
```

```
   13 - filter(("DATETIME">='190401010000' AND "DATETIME"<'190201010000'))
```

Finally, let's check whether query execution can be parallelized. And so it can. Notice the PX SELECTOR row sources in the query execution plan. This is a new feature of Oracle Database 12*c*. Oracle Database 12c is capable of executing UNION ALL branches in parallel. (See Concurrent Execution of Union All.)

```
alter table temperatures_1901 parallel 2;

select /*+ gather_plan_statistics */ * from
(
  select
    month,
    station_code,
    dense_rank() over (partition by month order by average) as rank
  from
  (
    select
      substr(datetime,1,4)||'/'||substr(datetime,5,2) as month,
      station_code,
      avg(temperature) as average
    from temperatures_v
    where datetime >= '190101010000' and datetime < '190501010000'
    group by
      substr(datetime,1,4)||'/'||substr(datetime,5,2),
      station_code
  )
)
pivot(max(station_code) for rank in (1, 2, 3))
order by month;

Plan hash value: 3783481314

-----------------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name              | Starts |    TQ  |IN-OUT| PQ Distrib  | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                   |      1 |        |      |             |     48 |00:00:00.50 |      64 |
|   1 |  PX COORDINATOR                          |                   |      1 |        |      |             |     48 |00:00:00.50 |      64 |
|   2 |   PX SEND QC (ORDER)                     | :TQ10002          |      0 |  Q1,02 | P->S | QC (ORDER)  |      0 |00:00:00.01 |       0 |
|   3 |    SORT GROUP BY                         |                   |      2 |  Q1,02 | PCWP |             |     48 |00:00:00.01 |       0 |
|   4 |     PX RECEIVE                           |                   |      2 |  Q1,02 | PCWP |             |     48 |00:00:00.01 |       0 |
|   5 |      PX SEND RANGE                       | :TQ10001          |      0 |  Q1,01 | P->P | RANGE       |      0 |00:00:00.01 |       0 |
|   6 |       HASH GROUP BY PIVOT                |                   |      2 |  Q1,01 | PCWP |             |     48 |00:00:00.02 |       0 |
|   7 |        VIEW                              |                   |      2 |  Q1,01 | PCWP |             |    288 |00:00:00.02 |       0 |
|   8 |         WINDOW SORT                      |                   |      2 |  Q1,01 | PCWP |             |    288 |00:00:00.02 |       0 |
|   9 |          HASH GROUP BY                   |                   |      2 |  Q1,01 | PCWP |             |    288 |00:00:00.02 |       0 |
|  10 |           PX RECEIVE                     |                   |      2 |  Q1,01 | PCWP |             |    288 |00:00:00.01 |       0 |
|  11 |            PX SEND HASH                  | :TQ10000          |      0 |  Q1,00 | P->P | HASH        |      0 |00:00:00.01 |       0 |
|  12 |             HASH GROUP BY                |                   |      2 |  Q1,00 | PCWP |             |    288 |00:00:00.92 |     368 |
|  13 |              VIEW                        | TEMPERATURES_V    |      2 |  Q1,00 | PCWP |             |  26266 |00:00:05.53 |     368 |
|  14 |               UNION-ALL                  |                   |      2 |  Q1,00 | PCWP |             |  26266 |00:00:02.79 |     368 |
|  15 |                PX BLOCK ITERATOR         |                   |      2 |  Q1,00 | PCWC |             |   6565 |00:00:01.12 |      90 |
|* 16 |                 EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1901 |      1 |  Q1,00 | PCWP |             |   6565 |00:00:00.98 |      90 |
|  17 |                PX SELECTOR               |                   |      2 |  Q1,00 | PCWP |             |   6565 |00:00:01.02 |      90 |
|* 18 |                 EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1902 |      2 |  Q1,00 | PCWP |             |   6565 |00:00:00.47 |      90 |
|  19 |                PX SELECTOR               |                   |      2 |  Q1,00 | PCWP |             |   6554 |00:00:00.49 |      85 |
|* 20 |                 EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1903 |      2 |  Q1,00 | PCWP |             |   6554 |00:00:00.44 |      85 |
|  21 |                PX SELECTOR               |                   |      2 |  Q1,00 | PCWP |             |   6582 |00:00:00.57 |      85 |
|* 22 |                 EXTERNAL TABLE ACCESS FULL | TEMPERATURES_1904 |      2 |  Q1,00 | PCWP |             |   6582 |00:00:00.52 |      85 |
-----------------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  16 - filter(("DATETIME">='190101010000' AND "DATETIME"<'190201010000'))
  18 - filter(("DATETIME">='190201010000' AND "DATETIME"<'190301010000'))
  20 - filter(("DATETIME">='190301010000' AND "DATETIME"<'190401010000'))
  22 - filter(("DATETIME">='190401010000' AND "DATETIME"<'190501010000'))
```

I predict that the time is soon coming when we will be able to store structured non-transactional data outside a transactional database management system while continuing to exploit the *entire* universe of indexing, partitioning, and clustering techniques as well as the *full* power of relational languages, *not only SQL*.