# SQL – the natural language for analysis

ORACLE WHITE PAPER | JUNE 2015

## Contents

.

## Overview

Analytics is a must-have component of every corporate data warehouse and big data project. Many of the early adopters of big data are managing the analysis of their data reservoirs through the use of specialized tools such as: MapReduce, Spark, Impala etc. This approach is locking data inside proprietary data silos, making cross-functional and cross-data store analysis either extremely difficult or completely impossible. IT teams are struggling to adapt complex silo-specific program code to support new and evolving analytical requirements. As a result, project costs are increasing and levels of risk within each project are also rising.

Many companies are searching for a single rich, robust, productive, standards driven language that can provide unified access over all types of data, drive rich sophisticated analysis, help constrain project costs and lower overall risk.

The objective of this paper is to explain why SQL is fast becoming the default language for data analytics and how it provides a mature and comprehensive framework for both data access, avoiding the creation of data silos, and supports a broad range of advanced analytical features.

## Introduction

Data processing has seen many changes and significant technological advances over the last forty years. However, there has been one technology, one capability that has endured and evolved: the Structured Query Language or SQL. Many other technologies have come and gone but SQL has been a constant. In fact, SQL has not only been a constant, but it has also improved significantly over time.

What is it about SQL that is so compelling? What has made it the most successful language? SQL's enduring success is the result of a number of important and unique factors:

- Powerful framework
- Transparent optimization
- Continuous evolution
- Standards based

The following sections of this paper will explore each of these key points that make SQL such a compelling language for data analysis.

## Powerful Framework

The reason for collecting data is because it needs to be interrogated. This interrogation process is typically framed around a set of data rather than a single data point or row of data. In 1970 E.F. Codd, while at IBM, established a set of mathematical rules that govern the construction and execution of a query. These rules are known as relational algebra (set theory) and consists of four primary operators:

- Projection
- Filter
- Join
- Aggregate

The following sections will explore these topics in more detail.

*1. Projection*

In the relational model, data is organized into rows and columns. When interrogating a set of data the first step is to determine which columns within the data set are of interest. These columns are referred to as projections of the total set of columns. Projections are specified in the first part of a SQL query's framework: the SELECT clause.

In the example below the set is employee data stored in a table called EMP. The required columns can be 'selected' from this data set:

```
SELECT ename, job, hiredate
FROM emp;
```

SQL also supports the concept of extended projections. This is where new data columns are created within the result set using arithmetic operations. For example, taking the columns `SAL` and `COMM` it is possible to compute the rate of commission by dividing `SAL` by `COMM` to create a new column `COMM_RATE` :

```
SELECT
 ename,
 sal,
 comm,
 comm/sal*100 AS COMM_RATE
FROM emp;
```

The SELECT clause can also include all the columns rather than simply a subset. There is a simple SQL syntax for selecting all columns within a set.

```
 SELECT * FROM emp;
```

The Oracle Database transparently uses its metadata layer to establish the list of all column names associated with the set and then internally expands the query statement to include all the relevant column names.

The Oracle Database has a comprehensive metadata layer that supports the discovery of columns across a wide range of data sets: not only relational tables/views, but also XML documents, JSON documents, spatial objects, image-style objects (BLOBs and CLOBs), semantic networks etc. Moreover, Oracle extends this metadata model beyond the Oracle Database so that it can encompass data stored in other repositories: NoSQL databases, JSON documents, and files stored on HDFS. This comprehensive metadata layer makes it very easy for developers to create SQL statements containing column projections based around a wide range of data types and data sources.

This combination of sophistication metadata and automatic statement expansion is a core part of SQL yet it is missing in many data analysis languages and this requires developers to add additional bespoke code to cope with these basic requirements.

*2. Filters*

The next stage within the query framework is to specify the rows that are of interest. Since these cannot be identified in the same way as columns, i.e. using names, a different approach is used for row-based projections. This approach requires the use of filters that describe the attributes associated with row sets that are of interest. Within the SQL language row filtering is part of the `WHERE` clause syntax (these filters are often called predicates in relational terminology). The developer can define specific filtering criteria that describe the rows to be included in the result set. For example using the employee data set it is possible to extend the previous examples to limit the set of rows returned by a query to just those associated with the job type 'CLERK' :

```
SELECT
 ename,
 job,
 hiredate
FROM emp
WHERE job='CLERK';
```

The SQL language supports a wide range of filtering comparison operators to identify specific rows, including: testing for equal to, not equal to, greater than, greater than or equal to, less than and less than or equal to. It is possible to check for a range between and including two values, for example to find all employees who have a salary range between $10,000 and $25,000:

```
SELECT
 ename,
 job,
 hiredate
FROM emp
WHERE sal BETWEEN 10000 AND 25000;
```

String searches are supported using wildcard symbols or within a list of given values. SQL can also check for the presence of null[1] values and either include or exclude them. This is a very powerful feature and is linked to the concept of joins (see following section).

Sometimes it might be necessary to exclude certain data from a result set. Using the prior example, it possible to return all jobs excluding clerks using the inequality operator, !=, as shown here:

```
SELECT
 ename,
 job,
 hiredate
FROM emp
WHERE job != 'CLERK';
```

The above query retrieves a list of employees except those that work as clerks. It is possible to construct a query that contains multiple filters to support more complex business requirements. The `WHERE` clause can be extended using the `AND` and `OR` operators. If two conditions are combined via the AND operator then both conditions must evaluate to true for a row to be included in the result, as shown below:

```
SELECT
 ename,
 job,
 hiredate
FROM emp
WHERE sal BETWEEN 10000 AND 25000
      AND job != 'CLERK';
```

1 Null is used to indicate that a data value does not exist. This is an important element of Codd's series of rules for relational algebra. It supports the concept of "*missing*" or "*not-applicable*" data points. For more information see the NULLs section in the SQL Language Reference

The above example returns a result set containing rows where the salary is between 10,000 and 25,000 and the job is not equal to 'CLERK'. Using the OR operator, only one of the conditions needs to return a true such that our query would return all the rows where the salary is between 10,000 and 25,000 or the job is not equal to 'CLERK'. This means that clerks earning between 10,000 and 25,000 will be included in the result set.

The SQL language has a very rich set of filtering techniques that are both simple to implement and to amend as requirements evolve over time.

*3. Joins*

Most query operations require the use of at least two data sets and this necessitates a "*join*" operation. At a simplistic level, a join is used to combine the fields within two or more data sets in a single query, based on common logical relationships between those various data sets. In our simple data example, suppose that there was a department data set in addition to the employee data set. A query might 'join' the department and employee data to return a single result combining data from both data sets. There are a number of ways that data sets can be joined:

- *Inner* - returns all rows where there is at least one match in both tables
- *Full Outer* - returns all rows from both tables, with matching rows from both sides where available. If there is no match, missing side will contain null.
- *Outer left* - returns all rows from left table, and matched rows from right table
- *Outer right* - returns all rows from right table, and matched rows from left table
- *Cross* - returns a Cartesian product of source sets, i.e. all rows from left table for each row in the right table

The process of defining and executing a SQL join is simple. The required type of join is implemented using the ANSI standard syntax (more on this towards the end of this paper) in the FROM clause:

```
SELECT
 d.deptno,
 d.dname,
 e.empno,
 e.ename
FROM dept d
     INNER JOIN
     emp e
     ON (e.deptno = d.deptno);
```

Note that there is nothing in this SQL query that describes the actual join process for the two data sets. In many of the procedural languages that are used with big data, the process of joining two sets of data is complicated by the need to explicitly code each join structure along with the join algorithm. The level of complication builds as additional data sets are added to the query; different join operations are required for combinations of data sets such as taking into account missing values within data sets.

The complexity of the join process can quickly escalate when the other join patterns need to be considered. For example when we join the two tables EMP and DEPT there could be departments that contains no employees. Using an inner join, the departments with zero employees are not be returned. If a query needs to return a count of the number of employees in every department, including the 'empty' departments containing zero employees, then an outer join is required as shown here:

```
SELECT
 d.deptno,
 count(e.empno)
```

```
FROM dept d
     LEFT OUTER JOIN
     emp e
     ON (e.deptno = d.deptno)
GROUP BY d.deptno;
```

It is clear that SQL's join code is easily readable – and code-able. The developer only specifies the semantic join condition and leaves the processing details - such as the order of the joins - to the SQL engine.

*4. Aggregate*

Aggregation is an important step in the process of analyzing data sets. Most operational, strategic and discovery-led queries rely on summarizing detailed level data. Industry analysts often state that up to 90% of all reports contain some level of aggregate information.

The types of aggregation applied to a data set can vary from simple counts to sums to moving averages to statistical analysis such as standard deviations. Therefore, the ability to simply and efficiently aggregate data is a key requirement for any analytical data language.

Procedural languages, such as the Java based MapReduce, are more than capable of taking a data set and aggregating it, or reducing it, to provide a summary result set. The approach is adequate for most simple data discovery style queries, i.e. those that include basic counts. However, adding more complex aggregation requirements quickly increases the amount of code required to manage the computations.

SQL has a rich set of data aggregation capabilities that make it easy to work on all rows in a set. For example, it is possible to sum all rows within a single column as follows:

```
SELECT SUM(sal) AS total_salary
FROM emp;
```

It is easy to extend this query to accommodate new requirements such as a count of the number of employees and the average salary:

```
SELECT
   COUNT(empno) AS no_of_employees,
   SUM(sal) AS total_salary,
   AVG(sal) As average_salary
FROM emp;
```

Taking these examples even further, it is a simple step to group rows into specific categories using the **GROUP BY** clause. The aggregate functions and **GROUP BY** clause group can be used to group the data and then apply the specific aggregate function(s) to count the number of employees, sum the salary and calculate the average salary in each department within a single query as shown below:

```
SELECT
   deptno,
   COUNT(empno) AS no_of_employees,
   SUM(sal) AS total_salary,
   AVG(sal) AS average_salary
FROM emp
GROUP BY deptno;
```

The ANSI SQL:2003 standard (more on this towards the end of this paper) extended the process of aggregating data by introducing the concept of analytic functions. These functions divide a data set into groups of rows called partitions making it is possible to calculate aggregates for each partition and for rows within each partition. The use of additional keywords define the how analytical functions, such as average, sum, min, max etc., will be evaluated within each partition. Using the previous example, the statement below creates analytical reporting totals such as: total salary within each department, moving average salary within each department and average salary:

```
    SELECT
       d.deptno,
       d.dname,
       e.ename,
       e.sal AS sal,
       SUM(e.sal) OVER (ORDER BY e.deptno)) AS dept_sal,
       ROUND(AVG(e.sal) OVER (PARTITION BY e.deptno ORDER BY e.empno)) AS moving_avg_sal,
       ROUND(AVG(e.sal) OVER (ORDER BY e.deptno)) AS avg_dept_sal
   FROM dept d
       LEFT OUTER JOIN
       emp e
       ON (e.deptno = d.deptno);
```

Compare the simplicity of the above SQL code for computing a moving average with the equivalent MapReduce code posted by Cloudera on Github as part of the blog post “Simple Moving Average, Secondary Sort, and MapReduce”[2]. This “simple” example consists of twelve java program files to perform the moving average calculation, manage the job parameters and specify the framework for the associated workflow. Making changes to the calculations as business requirements evolve will be a significant challenge given the amount of code within this “simple” project.

SQL offers an additional ability to apply restrictions using columns that are returned as part of the result set. The `HAVING` clause filters results based on calculations in the `SELECT` clause and/or aggregations derived from the `GROUP BY` processing, in the same way that `WHERE` filtering clause is applied, for example:

```
    SELECT
       deptno AS department,s
       COUNT(empno) AS no_employees,
       AVG(sal) AS avg_sal,
       SUM(sal) AS tot_sal
     FROM emp
     GROUP BY deptno
     HAVING avg(sal) > 2500;
```

Using SQL, developers and DBAs can leverage simple, convenient and efficient data aggregation techniques that require significantly less program code compared to using other analytical programming languages. The simplicity provided by SQL makes it easier and faster to construct, manage and maintain application code and incorporate new business requirements.

## Simplified Optimization

SQL is a declarative language. The declarative approach allows the developer to simply build the structure and elements of the required program in a way that expresses just the logic of the computation without having to describe the actual control flow. In simple terms it describes “what” the program, or query, should accomplish rather than describing “how” to actually accomplish the desired outcome. The “how” part of the processing is managed transparently by the underlying platform. This contrasts with the procedural approach to programming, which aims to break down a specific task into a collection of variables, procedures, functions, corresponding data structures and

2 http://blog.cloudera.com/blog/2011/04/simple-moving-average-secondary-sort-and-mapreduce-part-3/

conditional operators. It uses a list of instructions, wrapped up in procedures and subroutines, to tell a computer what to do, step-by-step. Many of today's big data languages adopt this procedural approach.

A simple example of returning a sorted list of employees, highlights the key differences between SQL's declarative-based code and the more workflow-orientated procedural languages such as Java:

**SQL's declarative approach**

```
SELECT
  empno,
  ename,
  hiredate
FROM emp
ORDER BY empno, hiredate, ename
```

**Java's procedural approach**[3]

```
public class Employee implements Comparable<Employee>{
    private int empno;
    private String ename;
    private int hiredate;

    public Employee(int empno, String name, int hiredate) {
        this.empno=empno;
        this.ename=ename;
        this.hiredate=hiredate;
    }

// Sort in descending order by Employee name. If name is same then sort in
descending order by age.
// If age is same then sort in descending order by Empid

    public int compareTo(Employee o){
                if (o.ename.equals(this.ename))
                        if (o.hiredate-this.hiredate == 0)
                                return o.empno - this.empno;
                        else return o.hiredate-this.hiredate;
                else return o.ename.compareTo(this.ename);
        }
      public int getHiredate(){
                return this.hiredate;
        }
      public int getEmpNo(){
                return this.empno;
        }
      public String getEname(){
                return this.ename;
        }
}

public class TestEmployeeSort {

        public static void main(String[] args) {
                List coll = Util.getEmployees();
                Collections.sort(coll); // sort method
                printList(coll);
                System.out.println("---------------------");
                Collections.sort(coll,new SortEmployeeByName());
                printList(coll);
        }

        private static void printList(List<Employee> list) {
                System.out.println("EmpId\tName\tAge");
                for (Employee e: list) {
                        System.out.println(e.getEmpId() + "\t" +
e.getName() + "\t" + e.getAge());
                }
        }
}

public class SortEmployeeByName implements Comparator<Employee>{
        public int compare(Employee o1, Employee o2){
                return o1.getName().compareTo(o2.getName());
        }
}
```

Many big data languages are procedural in their approach where the developer is responsible for implementing optimization techniques to support their workload. Each developer optimizes their code/process in complete isolation from the overall workload and while this granular level of control provides tremendous flexibility, it does create conflicts when there are multiple applications executing similar queries.

3 Example of how to sort a list in Java using Comparator

The initial focus on developer centric optimization techniques within big data projects is now giving way to a more platform centric approach as the complexity of analysis demanded by business users and data scientists continues to increase.

With declarative languages, such as SQL, the developer only has to understand the relationships between data entities, the final output and transformations needed to support the output. A declarative approach always creates code that is more compact, significantly less complex and much easier to maintain and enhance. The declarative code does not contain any instructions of how to process the data since it only expresses what the end result should be. The simplicity of a declarative language is one of the reasons why SQL is becoming the go-to language for big data analysis.

Of course, simplicity is only useful the SQL also scales and performs, and the Oracle Database implements a rich set of transparent SQL optimization techniques. The Oracle query optimizer is at the core of Oracle's database software and it determines the most efficient method for a SQL statement to access requested data. It attempts to generate the best execution plan[4] for a given SQL statement. During the process of determining the best plan, the optimizer examines multiple access methods, such as full table scan or index scans, and different join methods such as nested loops and hash joins before arriving at the "best execution plan". Due to the many internal statistics and tools within the database, the optimizer is usually in a better position than the developer to determine the best method of statement execution.

The Oracle Database also provides a powerful SQL parallel execution engine that can decide autonomously if and how parallel execution should be enabled for a query. The decisions on when to use parallel execution and the degree of parallelism chosen are both based on the resource requirements of a statement and the resources available at the time the query is submitted.

The Oracle Database Resource Manager (DBRM) allows the database administrator to prioritize workloads controlling access to resources. It can protect high priority tasks from being impacted by lower priority work by allocating CPU time to different jobs based on their priority.

By letting the database perform the optimizations at source, where the data is located, it is possible to share tuning improvements across a wide spectrum of front-end systems (BI reports, dashboards, applications etc.) each being optimized specifically for their own queries, but also optimized for the overall workload on the entire database system.

## Continuous Evolution

As an analytical language SQL has stayed relevant over the decades because, even though its core is grounded in the universal data processing techniques described above, it has been extended with new processing techniques, new calculations, and the ability to access new data types. Oracle has a long history of embedding sophisticated SQL-based analytics within the Oracle Database.

Window functions, which are now a key analytical feature in the analysis of big data, were first introduced in Oracle 8i (1999) and many developers use them to manage complex big data requirements. Oracle 10g (2003) introduced the SQL Model clause, which provides a spreadsheet-like what-if modeling framework aimed at business users allowing them to interact with both rows and columns within their data set to create new records within a given set.

4 An "execution plan" describes a recommended method of execution for a SQL statement. Each plan details the combination of steps that the Oracle Database must use to execute a SQL statement.

Oracle Database 12c includes a number of new big data related analytical functions which share common structure and syntax with existing SQL functions.

A good example of the evolution of analytical requirements is the need for approximate answers. In some cases the level of precision needed within an analytical query can be reduced – i.e. good enough is in fact the perfect answer. An approximate answer that is for example within 1% of the actual value can be sufficient. Oracle has already added this feature to the latest release of Database 12c by implementing HyperLogLog algorithms in SQL for 'approximate count distinct' operations.

As the range of types of data needed to support analysis has grown, SQL has been extended to reach out across an ever wider range of data sources, from relational tables/views to XML documents, JSON documents, Hive tables, HDFS files, spatial objects, image-style objects (BLOBs and CLOBs) and even semantic networks. This allows project teams to work with a single, rich analytical language across a wide range of data types and data sources, which reduces overall complexity and risk making it easier to incorporate enhancements and new requirements.

## Standards Based

One of the many challenges for big data projects is deciding which analytical language to make available to developers. In a recent report, one major consultancy organization[5] stated that project teams need to think seriously about standards for analytics.

Many of these open source big data languages are not bound by a set industry standards and this creates challenges when trying to integrate queries across multiple data reservoirs. Each language supports its own syntax, set of analytical functions and data types. The lack of consistency across languages typically results in large-scale data shipping between platforms because specific analytical functions are missing on one or other of the source platforms. Developers and business users find it difficult to work with features because of a lack of consistent naming, syntax and processing conventions.

At present there is a simple and poignant example of the implications of not following industry standards. The first wave of big data projects invested heavily in coding with the two-stage MapReduce procedural framework. This has now been superseded by the Apache Spark project, which aims to offer improved performance. However, Spark's framework is fundamentally different and requires project teams to rewrite existing MapReduce code.

The evolution of new processing languages in new big data platforms stands in stark contrast to the evolution of SQL. In 1986, SQL became a standard of the American National Standards Institute (ANSI) and since then it has advanced to its current iteration, ANSI 2011. This standardization has two major benefits:

First, the standard provides a high degree of application portability across different database systems without major code changes. In the field of data warehousing, BI tools are able to effectively support multiple types of SQL databases in a straightforward manner.

Second, the SQL standard has ensured continuity in application development. A SQL statement written thirty years ago continues to run today, without any modification to the SQL code. What is the key difference in today's SQL statement? The SQL now executes much faster, since the database transparently optimizes this SQL statement to take advantage of the latest performance enhancements.

5 Deloitte Analytics: Analytics Trends 2015: A Below-the-Surface Look.

Some of the big data languages have attempted to become more SQL-like and adopted specific elements of the ANSI standard. Many of these languages have adopted some or all of the SQL 92 standard. However, this is still not providing the level of portability that most project teams expect. A good example of this is Hive, which is being displaced by newer SQL implementations such as Impala. Both claim to adhere to the some of the earlier versions of the ANSI standards. Despite this, Impala does not support many of the most common HiveQL features such as aggregate functions, lateral views, multiple `DISTINCT` clauses per query etc. These differences make it extremely difficult to move code from one project to another without having to invest significant resources in recoding application logic. A lack of adherence to standards increases the overall level of risk within a project.

Since the publication of the SQL 92 standard a significant number of additional data warehouse features have been added to support more sophisticated analysis: SQL 99 introduced additional multi-dimensional aggregation features such as `ROLLUP`, `CUBE`, and `GROUPING SETS`; SQL 2003 added analytical windows functions and OLAP capabilities; SQL 2011 added support for temporal data types and analytics.

Databases that implement industry-standard SQL benefit their customers because industry-standard SQL maximizes the re-use and sharing of analytic processes, and minimizes the risks of those processes becoming obsolete.

## Why Oracle SQL?

Oracle's analytical SQL fully supports the ANSI SQL 2011 standard and also includes newer functions such as SQL pattern-matching and approximate count distinct that are being incorporated into the next iteration of the standard. This ensures broad support for these features and rapid adoption of newly introduced functionality across applications and tools – both from Oracle's partner network and other independent software vendors. Oracle is continuously working with its many partners to assist them in exploiting its expanding library of analytic functions. Already many independent software vendors have integrated support for the new Database 12c in-database analytic functions into their products.

The release of Oracle's Exadata platform delivered huge performance gains even when moving existing SQL code and running it unchanged, even code written in the 1980's. The storage level performance optimizations such as storage indexes, hybrid columnar compression and smart scans, which were incorporated into the Exadata Storage Server, are completely transparent to the SQL code. This allowed every application, every SQL statement to run unchanged on this platform and reap the benefits of running on an engineered and optimized platform.

As part of its on-going investment Oracle has released in-memory processing to further improve the performance of SQL queries. The management and lifecycle of the in-memory data structures and the use of different types of compression depending on the type of operations (OLTP-centric vs. Analytics-centric) are transparent to the queries. Developers and DBAs are not forced to rewrite their application code to leverage this feature. As with Exadata, all existing queries can automatically inherit the benefits.

Oracle SQL allows project teams to work with a single, rich analytical language, which reduces overall complexity, avoids the creation of data silos and makes it easier for IT teams to incorporate enhancements and new requirements from business teams.

## Conclusion

*SQL has emerged as the de facto language for data analysis simply because it is technically superior and the significant benefits that the language provides.*

IT teams are facing new challenges as increasing amounts of granular data is being collected, applications and dashboards need to be updated with the latest information in real-time, and business requirements continually evolve minute by minute. To help resolve these challenges, IT teams need a flexible powerful framework for data analysis.

The flexibility and power of SQL make it a vital tool for all data analysis projects and an ever-growing number of IT teams are using SQL as the go-to language for analysis because of a number of important and unique factors:

- It provides a powerful framework
- Processing optimizations are completely transparent
- It continuously evolves to meet today's demanding requirements
- Adherence to international ANSI standards

Already, many companies are using Oracle and SQL to drive sophisticated analysis across all their data sets as part of an agile development ecosystem. This is because it provides a mature and comprehensive framework for both data access, which stops the creation of data silos, and rich analysis that avoids shipping data to specialized processing engines.

Oracle's SQL provides business users and big data developers with a simplified way to support the most complex data discovery and business intelligence reporting requirements. Its support for the very latest industry standards, including ANSI 2011, industry-leading optimizations, commitment to continuous innovation has ensured that SQL is now the default language for analytics across all types of data and data sources.

Overall, Oracle's SQL helps project teams reduce overall project costs and reduce risk. The analysis that it delivers allows organizations to explore product innovations, seek out new markets and, most importantly, it offers the potential to build significant competitive advantage.

## Further Reading

The following Oracle Database features are referenced in the text:

1. Database SQL Language Reference - Oracle and Standard SQL
2. Oracle Analytical SQL Features and Functions - a compelling array of analytical features and functions that are accessible through SQL
3. Oracle Statistical Functions - eliminate movement and staging to external systems to perform statistical analysis.
4. Oracle Database 12c Query Optimization - providing innovation in plan execution and stability.

5. Oracle Big Data SQL - one fast query, on all your data
6. Oracle Database In-Memory - supporting real-time analytics, business intelligence, and reports.

The following Oracle white papers and data sheets are essential reading:

1. Effective Resource Management Using Oracle Database Resource Manager
2. Best Practices for Gathering Optimizer Statistics with Oracle Database 12c
3. Parallel Execution and Workload Management for an Operational Data Warehouse
4. Manageability with Oracle Database 12c
5. Best Practices for Workload Management of a Data Warehouse on the Oracle Exadata Database Machine

You will find links to the above papers, and more, on the “Oracle Data Warehousing Best Practice” web page hosted on the Oracle Technology Network:

http://www.oracle.com/technetwork/database/bi-datawarehousing/dbbi-tech-info-best-prac-092320.html