# Configure users for ms sql

<https://www.papercut.com/support/resources/manuals/ng-mf/common/topics/ext-db-specific-ms-sql-express.html>

Import adventure work database for ms sql`

# SQL datetime, date, time LITERAL

All the literal value must be put in quotes

ISO format

Date: 'yyyy-mm-dd'

Time: 'hh:mm:ss.sss'

DateTime: 'yyyy-mm-dd hh:mm:ss.sss' or 'yyyy-mm-dd**T**hh:mm:ss.sss'

If you omit the time then 'yyyy-mm-dd' (for datet time format) means hh:mm:ss.sss = 00:00:00.000

(Note that 'yyyymmdd' can be used for date, but cannot be used with time like

'yyyymmdd hh:mm:ss.sss' or 'yyyymmdd**T**hh:mm:ss.sss'

ODBC format:

Dates: {d 'yyyy-mm-dd'} {d '2001-12-31'}

Times: {t 'hh:mm:ss'}

Timestamps: {ts 'yyyy-mm-dd hh:mm:ss'} {ts '2001-12-31 00:00:00'}

# For each Statement, one ResultSet at a time: Creating a new ResultSet will close the current ResultSet

From Java doc:

By default, only one ResultSet object per Statement object can be open at the same time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All execution methods in the Statement interface implicitly close a statment's current ResultSet object if an open one exists.

# Order of execution: FROM, JOIN > WHERE > GROUP BY, HAVING >SELECT, WINDOW, DISTINCT, ORDER BY

**1. FROM and JOINs**

The **FROM** clause, and subsequent **JOIN**s are first executed to determine the total working set of data that is being queried. This includes subqueries in this clause, and can cause temporary tables to be created under the hood containing all the columns and rows of the tables being joined.

**2. WHERE**

Once we have the total working set of data, the first-pass **WHERE** constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. *Each of the constraints can only access columns directly from the tables requested in the****FROM****clause.*

*Aliases in the****SELECT****part of the query are not accessible in most databases since they may include expressions dependent on parts of the query that have not yet executed.*

**3. GROUP BY**

The remaining rows after the **WHERE** constraints are applied are then grouped based on common values in the column specified in the **GROUP BY** clause. As a result of the grouping, there will only be as many rows as there are unique values in that column. Implicitly, this means that you should only need to use this when you have aggregate functions in your query.

*Like the****WHERE****clause, aliases are not yet computed so they are not accessible from this step.*

**4. HAVING**

If the query has a **GROUP BY** clause, then the constraints in the **HAVING** clause are then applied to the grouped rows, discard the grouped rows that don't satisfy the constraint.

*Like the****WHERE****clause, aliases are not yet computed so they are not accessible from this step.*

**5. SELECT**

Any expressions in the **SELECT** part of the query are finally computed.

Windows functions are computed at this step.

**6. DISTINCT**

Of the remaining rows, rows with duplicate values in the column marked as **DISTINCT** will be discarded.

**7. ORDER BY**

If an order is specified by the **ORDER BY** clause, the rows are then sorted by the specified data in either ascending or descending order. Since all the expressions in the **SELECT** part of the query have been computed, you can reference aliases in this clause.

**8. LIMIT / OFFSET**

Finally, the rows that fall outside the range specified by the **LIMIT** and **OFFSET** are discarded, leaving the final set of rows to be returned from the query.

# Alias in Select cannot appear in Where, Group By, Having

Alias in Select (e.g. select substring(name, 1, 4) as ALIAS) are generated/computed almost the last in the execution order while WHERE, GROUP BY, HAVING are computed beforehand so the alias cannot appear in Where, Group by, Having.

Exp: The two followings are meant to do the same thing but (1) doesn’t compile.

1. ~~select substring(name, 1, 4) as Alias from employee where Alias = ‘Mike’~~
2. select substring(name, 1, 4) as Alias from employee where substring(name, 1, 4) = ‘Mike’

# Group by

## Careful: Aggregate functions implies group by, which limits fields in select

Exp 1: “Select department\_name, count(employee\_name) From Employee” doesn’t compile since count() implies Group by nothing. And group by requires the fields in select to be included in Group by or put in aggregate functions. Here group by nothing means any field in Select must be in aggregate functions. And department\_name doesn’t satisfy this requirement.

Exp 2: “select count(department\_name), count(employee\_name) works fine.

## GROUP BY

1. GROUP BY can have more than 1 field.  
   Group By X: put all rows with same X value in a group

Group By X, Y: put all rows with same both X, Y value in a group

1. **If you use GROUP BY, then fields in SELECT, HAVING must be either in GROUP BY or in an AGGREGATE function.**
2. Having is used to limit the number of groups.

# Having

1. **WHERE doesn’t accept aggregate functions. Conditions with Aggregate functions can be used only in HAVING.**
2. Having doesn’t need to go with GROUP BY. Having without Group by is done by viewing all the rows as a single group. Having requires aggregate functions that are often applied on files in Group by, this is why Having often goes with Group by.
3. The fields in HAVING must be either in GROUP BY or in AGGREGATE FUNCTIONS.

# OVER (PARTITION BY … ORDER BY …) vs GROUP BY … ORDER BY …

Example: Consider the following “Employee” table

|  |  |  |  |
| --- | --- | --- | --- |
| Employee | | | |
| ID | EmployeeName | EmployeeAge | DepartmentName |
| 1 | Michael | 50 | Accounting |
| 2 | Frances | 60 | Accounting |
| 3 | Daisy | 20 | Accounting |
| 4 | Dennis | 40 | Software |
| 5 | Andrei | 25 | Software |
| 6 | Carol | 25 | Software |
| 7 | Peter | 40 | Human resource |
| 8 | Larry | 35 | Human resource |
| 9 | Robert | 35 | Human resource |
| 10 | Leonard | 22 | Marketing |
| 11 | Andrew | 24 | Marketing |

**Group by:**

**Exp1**: count the employees of each department that has more than 2 employees

Select DepartmentName, count(EmployeeName) as Count

From Employee

Group by DeparmentName

Having count(EmployeeName) > 2

|  |  |
| --- | --- |
| Group by | |
| DepartmentName | Count |
| Accounting | 3 |
| Software | 3 |
| Human resource | 3 |

(Note: it doesn’t compile if use “where” instead of “having”. **Aggregate functions or Alias are not allowed in Where**

~~Select DepartmentName, count(EmployeeName) as tempCount~~

~~From Employee~~

**~~Where tempCount > 2~~**

~~Group by DeparmentName~~)

Exp2: Counting number of employees older than 25 for each department

Select DepartmentName, count(EmployeeAge) as X

From Employee

where EmployeeAge > 25

Group by DepartmentName

|  |  |
| --- | --- |
| Group by | |
| DepartmentName | Count |
| Accounting | 2 |
| Software | 1 |
| Human resource | 3 |

(Note: it doesn’t compile if use “Having” instead of “Where” since EmployeeAge must be in Group By

~~Select DepartmentName, count(EmployeeAge) as X~~

~~From Employee~~

~~Group by DepartmentName~~

~~Having EmployeeAge > 25~~)

**Over partition by:**

**Exp1**: count the employees of each department that has more than 2 employees

The following doesn’t compile since X is an alias that cannot be in where

~~Select EmployeeName, DepartmentName, count(EmployeeName) over (partition by DepartmentName) as x~~

~~from Employee~~

~~where X > 2~~

Exp2: Counting number of employees older than 25 for each department

Select EmployeeName, DepartmentName, count(EmployeeName) over (partition by DepartmentName) as Count

From Employee

Where EmployeeAge > 25

|  |  |  |  |
| --- | --- | --- | --- |
| Over partition by | | | |
| EmployeeName | EmployeeAge | DepartmentName | Count |
| Michael | 50 | Accounting | 2 |
| Frances | 60 | Accounting | 2 |
| Dennis | 40 | Software | 1 |
| Peter | 40 | Human resource | 3 |
| Larry | 35 | Human resource | 3 |
| Robert | 35 | Human resource | 3 |

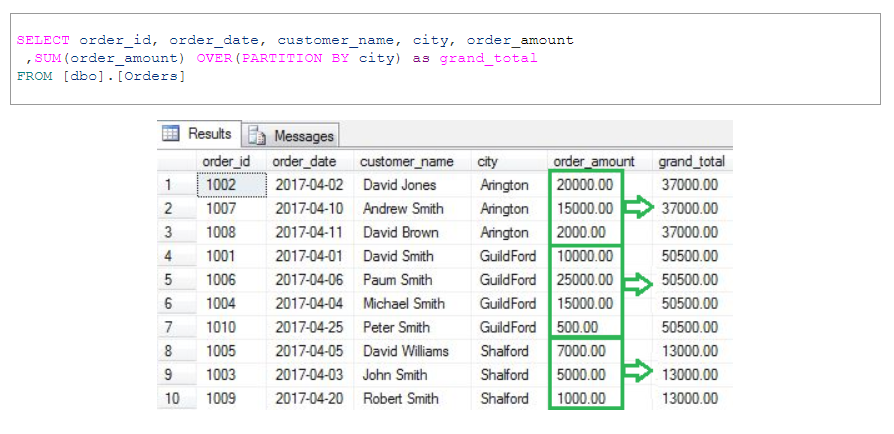
The most important difference is “Aggregate vs Analytic”.

In the above example, for Group By, only groups plus some properties of the groups are viewed. For Over partition by, individual elements of groups and properties of groups are displayed so there must be repetition.

Viewing group is called aggregate while viewing elements is called analytic.

# Analytic functions

## Why is it called WINDOW functions?



## Common analytic functions

* **All the aggregate functions** can be used as analytic functions: COUNT, MAX, MIN, SUM, AVG, STDEV, STDEVP
* First\_value(), last\_value()
* Row\_number()
* Rank()

## Some specific queries with analytic functions

### Analytic functions with multiple tables: join them

Exp:

Department(departmentID, departmentName, departmentAddress)

Employee(employeeID, employeeName, employeeGender, departmentID)

Equipment(equipmentID, equipmentName, equipmentPrice, departmentID)

**For each department, list the first employee’s name and the highest price of the department’s equipment**

**select** **distinct** dept.departmentID, dept.departmentName,

**first\_value**(emp.employeeName) **over** (**partition** **by** dept.departmentID **order** **by** emp.employeeID), **max**(eqm.equipmentPrice) **over** (**partition** **by** dept.departmentID)

**from** Department dept

**join** Employee emp **on** dept.departmentID = emp.departmentID

(code to create and populate the tables:

**CREATE** **TABLE** Department(departmentID **int** **PRIMARY** **key**, departmentName **varchar**(30), departmentAddress **varchar**(100))

**CREATE** **TABLE** Employee(employeeID **int** **PRIMARY** **key**, employeeName **varchar**(300), employeeGender **char**(1), departmentID **int** **foreign** **key** **references** Department(departmentID))

**CREATE** **TABLE** Equipment(equipmentID **int** **PRIMARY** **key**, equipmentName **varchar**(300), equipmentPrice **int**,

departmentID **int** **foreign** **key** **references** Department(departmentID))

**insert** **into** Department **values** (1, 'Software Dev', '101 T building'), (2, 'Accountance', '102 T building'), (3, 'Human Resource', '103 T building'), (4, 'Security', '104 T building'), (5, 'Planning', '105 T building')

**insert** **into** Employee **values** (1, 'Thang', 'M', 1), (2, 'Thu', 'F', 1), (3, 'Larry', 'M', 1), (4, 'Steven', 'M', 2), (5, 'Tamuko', 'F', 2), (6, 'Koki', 'F', 2), (7, 'Gate', 'M', 3), (8, 'Ellision', 'F', 3), (9, 'Jac Ma', 'M', 3), (10, 'Mark', 'M', 3), (11, 'Bezoff', 'F', 4), (12, 'Hillary', 'F', 4), (13, 'Trump', 'M', 4), (14, 'Biden', 'F', 4), (15, 'Washington', 'M', 5), (16, 'Warren', 'M', 5), (17, 'Tom', 'F', 5), (18, 'Peter', 'F', 5), (19, 'Hanks', 'M', 5), (20, 'Bush', 'F', 1), (21, 'Ivanka', 'M', 2), (22, 'Marilyn', 'M', 3), (23, 'Ellon', 'F', 4), (24, 'Cuban', 'F', 5)

**insert** **into** Equipment **values** (1, 'Computer', 1000, 1), (2, 'Printer', 510, 2), (3, 'Monitor', 130, 3), (4, 'Keyboard', 10, 4), (5, 'Mouse', 12, 5), (6, 'Table', 135, 1), (7, 'Chair', 35, 2), (8, 'Pen', 1, 3), (9, 'Notebook', 1, 4), (10, 'Eraser', 1, 5), (11, 'Server', 2000, 1), (12, 'Router', 105, 2), (13, 'Switch', 90, 3), (14, 'Thermal', 200, 4), (15, 'Car', 5600, 5), (16, 'Coffee maker', 93, 1), (17, 'Paper', 25, 2), (18, 'Cattridge', 147, 3), (19, 'Calendar', 4, 4), (20, 'Board', 29, 5)

**select** \* **from** Department

**select** \* **from** Employee

**select** \* **from** Equipment

**For** **each** department, list the **first** male employee **and** the most expensive equipment

**select** **distinct** dept.departmentID, dept.departmentName,   
**first\_value**(emp.employeeName) **over** (**partition** **by** dept.departmentID **order** **by** emp.employeeID), **max**(eqm.equipmentPrice) **over** (**partition** **by** dept.departmentID)

**from** Department dept

**join** Employee emp **on** dept.departmentID = emp.departmentID

**join** Equipment eqm **on** dept.departmentID = eqm.departmentID)

### When to use row\_number()

Consider still the example above,

Department(departmentID, departmentName, departmentAddress)

Employee(employeeID, employeeName, employeeGender, departmentID)

**For each department, view name of the first employee**

Don’t use row\_number():

**select** **distinct** d.departmentID, d.departmentName,

first\_value(e.employeeName) **over** (**partition** **by** d.departmentID **order** **by** e.employeeID)

**from** Department d

**join** Employee e **on** d.departmentID = e.departmentID

Use row\_number():

**select** \*

**from**

(**select** d.departmentID, d.departmentName, e.employeeName,

**row\_number**() **over** (**partition** **by** d.departmentID **order** **by** e.employeeID) **as** indexcol

**from** Department d

**join** Employee e **on** d.departmentID = e.departmentID) temptable

**where** indexcol = 1

**For each department, view ALL the information of the first employee**

If not using row\_number(), one has to do over partition by for each column.

Using row\_number() then one need to use it once to have all the columns.

**For each department, view ALL the information of the first employee, and the first equipment**

Consider still the example above with additional Equipment table

Department(departmentID, departmentName, departmentAddress)

Employee(employeeID, employeeName, employeeGender, departmentID)

Equipment(equipmentID, equipmentName, equipmentPrice, departmentID)

**select** \*

**from** Department

join (**select** \*, **row\_number**() **over** (**partition** **by** departmentID **order** **by** employeeID) **as** EmRank

from Employee) Em on Em.departmentID = Department.departmentID

join (**select** \*, **row\_number**() **over** (**partition** **by** departmentID **order** **by** equipmentID) **as** EqRank

from Equipment) Eq on Eq.departmentID = Department.departmentID

**where** EmRank = 1 & EqRank = 1

### For every group, list only some members

Exp: still with Employee table above, for each department, listing the oldest employee

Subquery:

Select DepartmentName, EmployeeName

From Employee

Where EmployeeAge = (Select MAX(EmployeeAge)

From Employee as tempEmployee

Where tempEmployee.DepartmentName = Employee.DepartmentName)

Over partition by

select \*

from (select DepartmentName, EmployeeName, EmployeeAge,

row\_number() over (partition by DepartmentName order by EmployeeAge DESC) as rank

from Employee) as extendedEmployee

where extendedEmployee.rank = 1

# with tableName as (select from where)

“with” must be followed by Select/Insert/Update command and it’s used only ONCE.

“with” cannot be followed by “with”. If want so, put multiple “with” in one command, separated by comma.

With name1 as (…), name2 as (…)

# Some other erros

### In FROM clause, a user-defined table must have a name

### “The column was specified multiple times”: columns with same names being called in Select

When joining two tables that have some same columns:

* If you just view it (select) then this doesn’t cause any problem. Columns with same names will be shown

**select** \*

**from** EQUIP\_MASTER em

**left** **join** STAT\_COND\_HIST sch **on** em.master\_num = sch.master\_num

* If you make the result as a table (which now has columns of same names) then accessing this table causes an error

**select** \* **from**

(**select** \*

**from** EQUIP\_MASTER em

**left** **join** STAT\_COND\_HIST sch **on** em.master\_num = sch.master\_num) temptable

# Join With First Matching Rows 3 methods: subquery, cross apply, row\_number

<http://andreyzavadskiy.com/2015/11/18/sql-join-with-first-matching-rows-choosing-the-best-approach/>

It’s a very old task for SQL developers to write a query that will join two tables and will pick only first matches from the second table to every row from the first table. In my case I need to combine client name from Clients table and his/her first phone number from Phones table.

After making some investigations I stopped at three different queries.

**1. Join with SELECT TOP 1 subquery**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | SELECT c.ClientName, ph.PhoneNumber  FROM Clients c  JOIN Phones ph ON c.ClientGuid = ph.ClientGuid    AND ph.PhoneNumber = (      SELECT TOP 1 p.PhoneNumber      FROM Phones p      WHERE p.ClientGuid = c.ClientGuid      ORDER BY p.PhonePriority      ); |

**2. Using CROSS APPLY operator**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT c.ClientName, ph.PhoneNumber  FROM Clients c  CROSS APPLY (    SELECT TOP 1 p.PhoneNumber    FROM Phones p    WHERE p.ClientGuid = c.ClientGuid    ORDER BY p.PhonePriority    ) ph; |

**3. Subquery with SQL Server Window function (ROW\_NUMBER)**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | SELECT c.ClientName, ph.PhoneNumber  FROM Clients c  JOIN (    SELECT p.ClientGuid, p.PhoneNumber,      ROW\_NUMBER() OVER (        PARTITION BY p.ClientGuid        ORDER BY p.PhonePriority        ) AS row\_num    FROM Phones p    ) ph    ON ph.ClientGuid = c.ClientGuid AND row\_num = 1; |

If any client doesn’t have a phone number, you need to make some slight modifications to the code above: change JOIN to LEFT JOIN, and CROSS APPLY to OUTER APPLY. In this case you’ll have a client name with corresponding NULL instead a phone number.

I won’t estimate the simplicity of the code and ease of understanding. The code that looks shorter might not be the most effective. We need to compare query costs and choose the least one.

And now SQL Server will show its magic. The percentage of each query costs are 50%, 49% and 1% (just look at the screenshot below).

[](http://andreyzavadskiy.com/wp-content/uploads/2015/11/Join-with-first-matching-rows.png)

So the most effective is the last query that uses a join with a ranking subquery (SQL Server creates a temporary table here). This query also operates with a minimum number of pages to retrieve the result. You can switch on the I/O statistics (run SET STATISTICS IO ON command) and look at Messages tab in SSMS. In my case I have the following output:  
(70347 row(s) affected)  
Table 'Phones'. Scan count 70713, logical reads 215349...  
Table 'Clients'. Scan count 5, logical reads 833...  
Table 'Worktable'. Scan count 0, logical reads 0...  
Table 'Worktable'. Scan count 0, logical reads 0...

(70347 row(s) affected)  
Table 'Phones'. Scan count 70708, logical reads 213139...  
Table 'Clients'. Scan count 1, logical reads 761...

(70347 row(s) affected)  
Table 'Phones'. Scan count 5, logical reads 2210...  
Table 'Clients'. Scan count 5, logical reads 833...  
Table 'Worktable'. Scan count 0, logical reads 0...

**Tip:** If you would use a LEFT JOIN in the last query, don’t place a “*row\_num = 1*” condition in the WHERE clause, only after JOIN … ON. If you place it in WHERE clause, SQL Server will make an left outer join, and then filter rows (all NULL values will be rejected here). So you will get an equivalent of inner join.

# JOIN, be it left join, outer join, inner join can multiply the number of records

# INDEX: cluster vs non-cluster

## Note 1

Index is a on-disk structure associated with a table that helps DBMS to ***fast*** retrieve the table data from the hard disk.

If a table is a book, then index is the index page that shows the page number for the information you need.

Each index is determined by a group of columns.

2 type of index for tables: clustered (physical index) and non-clustered index

## Note 2

An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

A table or view can contain the following types of indexes:

* Clustered
  + Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.
  + The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.
* Nonclustered
  + Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.
  + The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

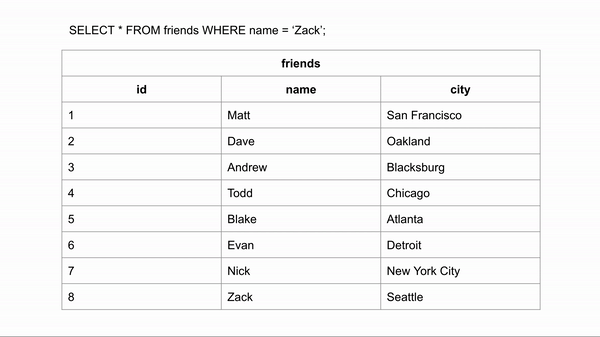
## Note 3

#### **What is Indexing?**

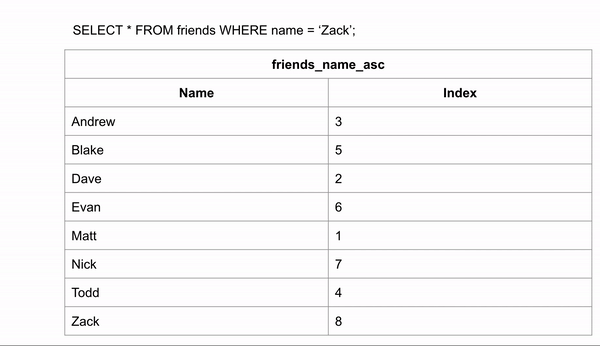
Indexing makes columns faster to query by creating pointers to where data is stored within a database.

Imagine you want to find a piece of information that is within a large database. To get this information out of the database the computer will look through every row until it finds it. If the data you are looking for is towards the very end, this query would take a long time to run.

Visualization for finding the last entry:



If the table was ordered alphabetically, searching for a name could happen a lot faster because we could skip looking for the data in certain rows. If we wanted to search for “Zack” and we know the data is in alphabetical order we could jump down to halfway through the data to see if Zack comes before or after that row. We could then half the remaining rows and make the same comparison.



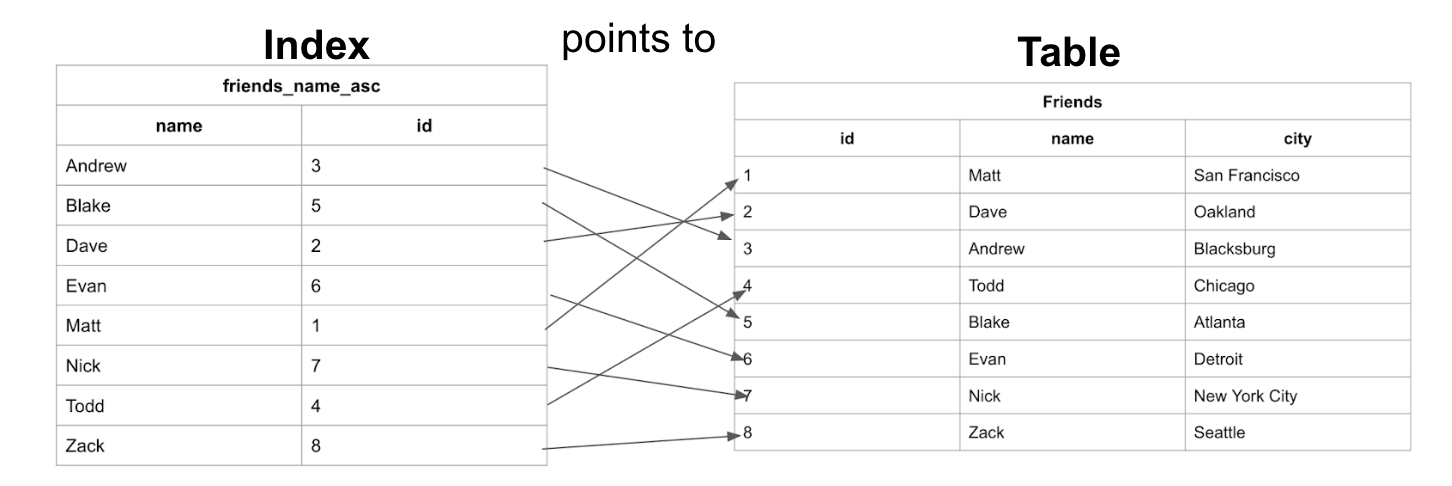
This took 3 comparisons to find the right answer instead of 8 in the unindexed data.

Indexes allow us to create sorted lists without having to create all new sorted tables, which would take up a lot of storage space.

#### **What Exactly is an Index?**

An index is a structure that holds the field the index is sorting and a pointer from each record to their corresponding record in the original table where the data is actually stored. Indexes are used in things like a contact list where the data may be physically stored in the order you add people’s contact information but it is easier to find people when listed out in alphabetical order.

Let’s look at the index from the previous example and see how it maps back to the original Friends table:



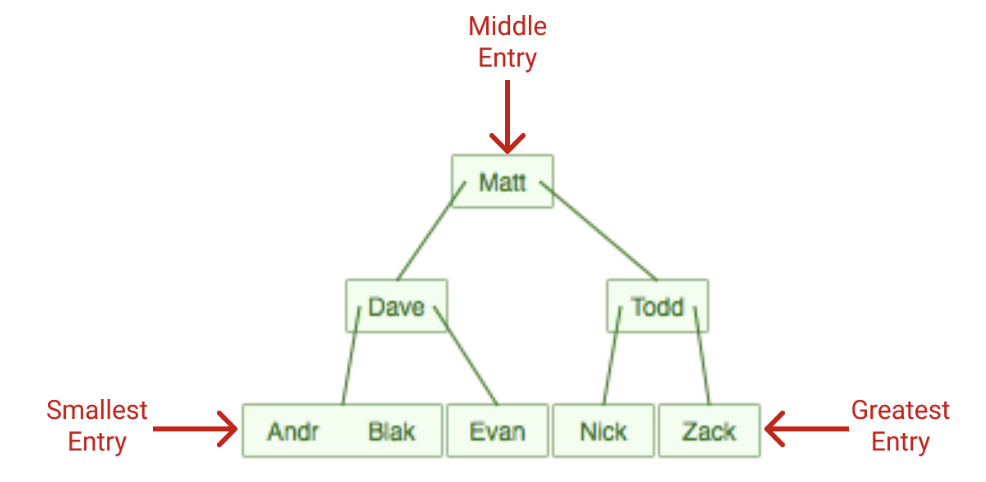
We can see here that the table has the data stored ordered by an incrementing id based on the order in which the data was added. And the Index has the names stored in alphabetical order.

#### **Types of Indexing**

There are two types of databases indexes:

1. **Clustered**
2. **Non-clustered**

Both clustered and non-clustered indexes are stored and searched as B-trees, a data structure similar to a [binary tree](https://en.wikipedia.org/wiki/Binary_tree). A [B-tree](https://en.wikipedia.org/wiki/B-tree) is a “self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.” Basically it creates a tree-like structure that sorts data for quick searching.



Here is a B-tree of the index we created. Our smallest entry is the leftmost entry and our largest is the rightmost entry. All queries would start at the top node and work their way down the tree, if the target entry is less than the current node the left path is followed, if greater the right path is followed. In our case it checked against Matt, then Todd, and then Zack.

To increase efficiency, many B-trees will limit the number of characters you can enter into an entry. The B-tree will do this on it’s own and does not require column data to be restricted. In the example above the B-tree below limits entries to 4 characters.

##### **Clustered Indexes**

Clustered indexes are the unique index per table that uses the primary key to organize the data that is within the table. The clustered index ensures that the primary key is stored in increasing order, which is also the order the table holds in memory.

* Clustered indexes do not have to be explicitly declared.
* Created when the table is created.
* Use the primary key sorted in ascending order.

#### **Creating Clustered Indexes**

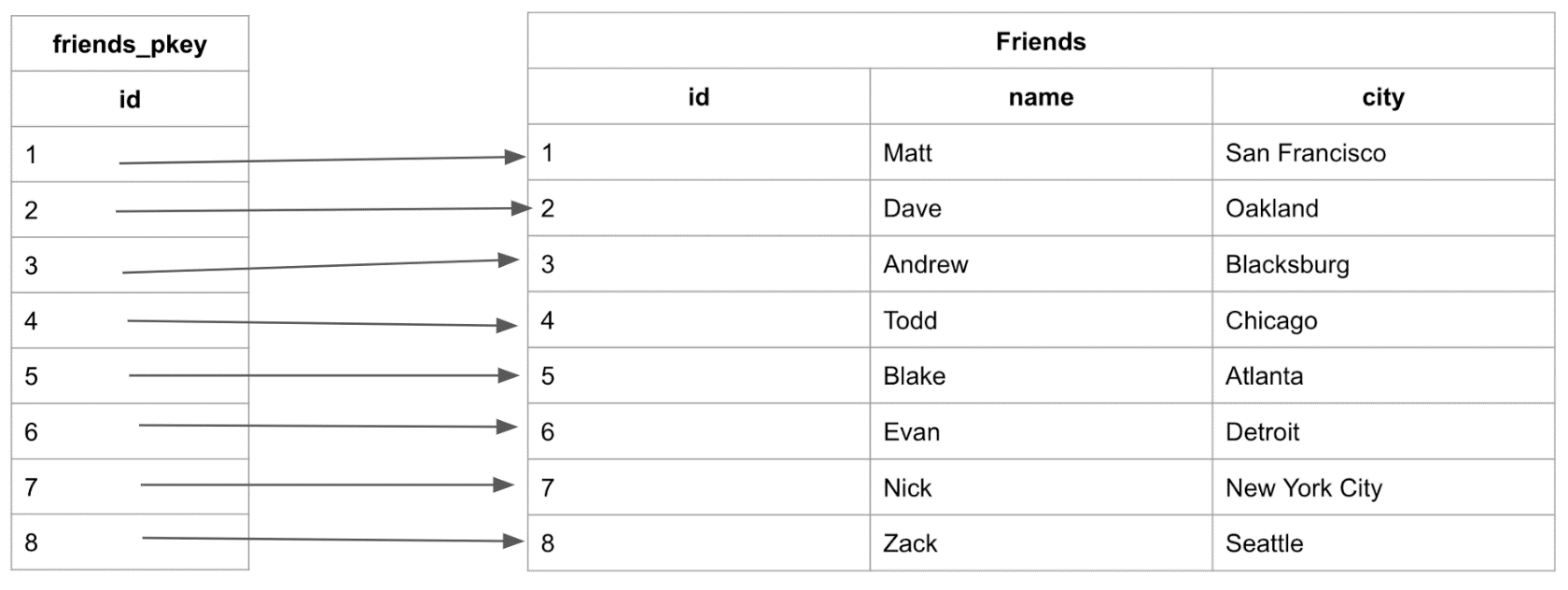
The clustered index will be automatically created when the primary key is defined:

**CREATE** **TABLE** friends (id INT **PRIMARY** **KEY**, name VARCHAR, city VARCHAR);

Once filled in, that table would look something like this:



The created table, “friends”, will have a clustered index automatically created, organized around the Primary Key “id” called “friends\_pkey”:

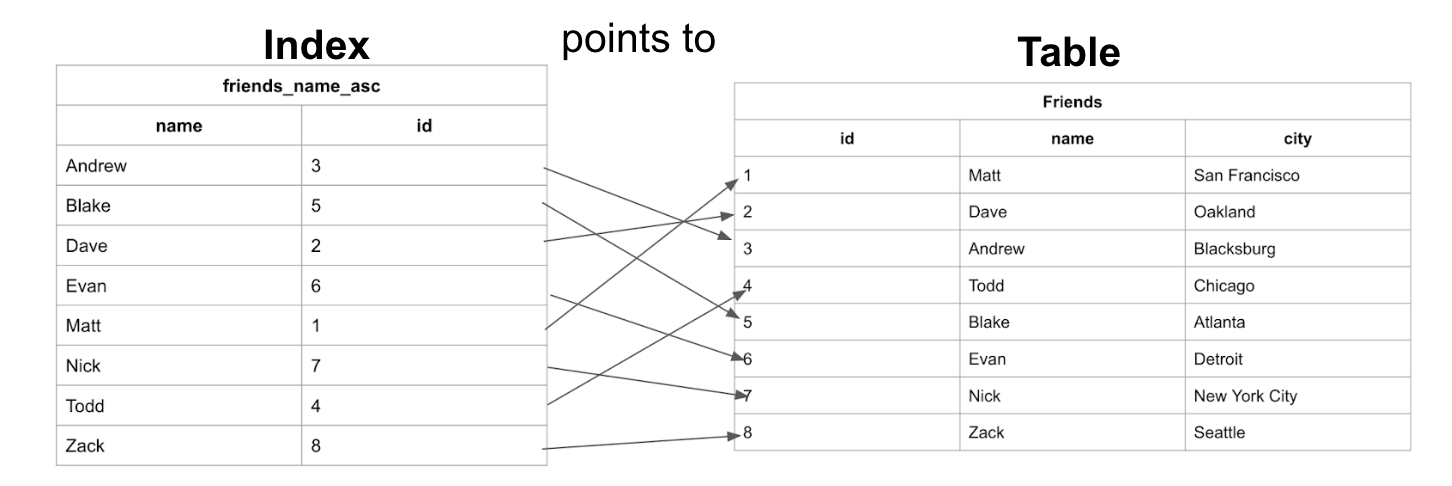


When searching the table by “id”, the ascending order of the column allows for optimal searches to be performed. Since the numbers are ordered, the search can navigate the B-tree allowing searches to happen in logarithmic time.

However, in order to search for the “name” or “city” in the table, we would have to look at every entry because these columns do not have an index. This is where non-clustered indexes become very useful.

##### **Non-Clustered Indexes**

Non-clustered indexes are sorted references for a specific field, from the main table, that hold pointers back to the original entries of the table. The first example we showed is an example of a non-clustered table:



They are used to increase the speed of queries on the table by creating columns that are more easily searchable. Non-clustered indexes can be created by data analysts/ developers after a table has been created and filled.

Note: Non-clustered indexes are **not** new tables. Non-clustered indexes hold the field that they are responsible for sorting and a pointer from each of those entries back to the full entry in the table.

You can think of these just like indexes in a book. The index points to the location in the book where you can find the data you are looking for.



Non-clustered indexes point to memory addresses instead of storing data themselves. This makes them slower to query than clustered indexes but typically much faster than a non-indexed column.

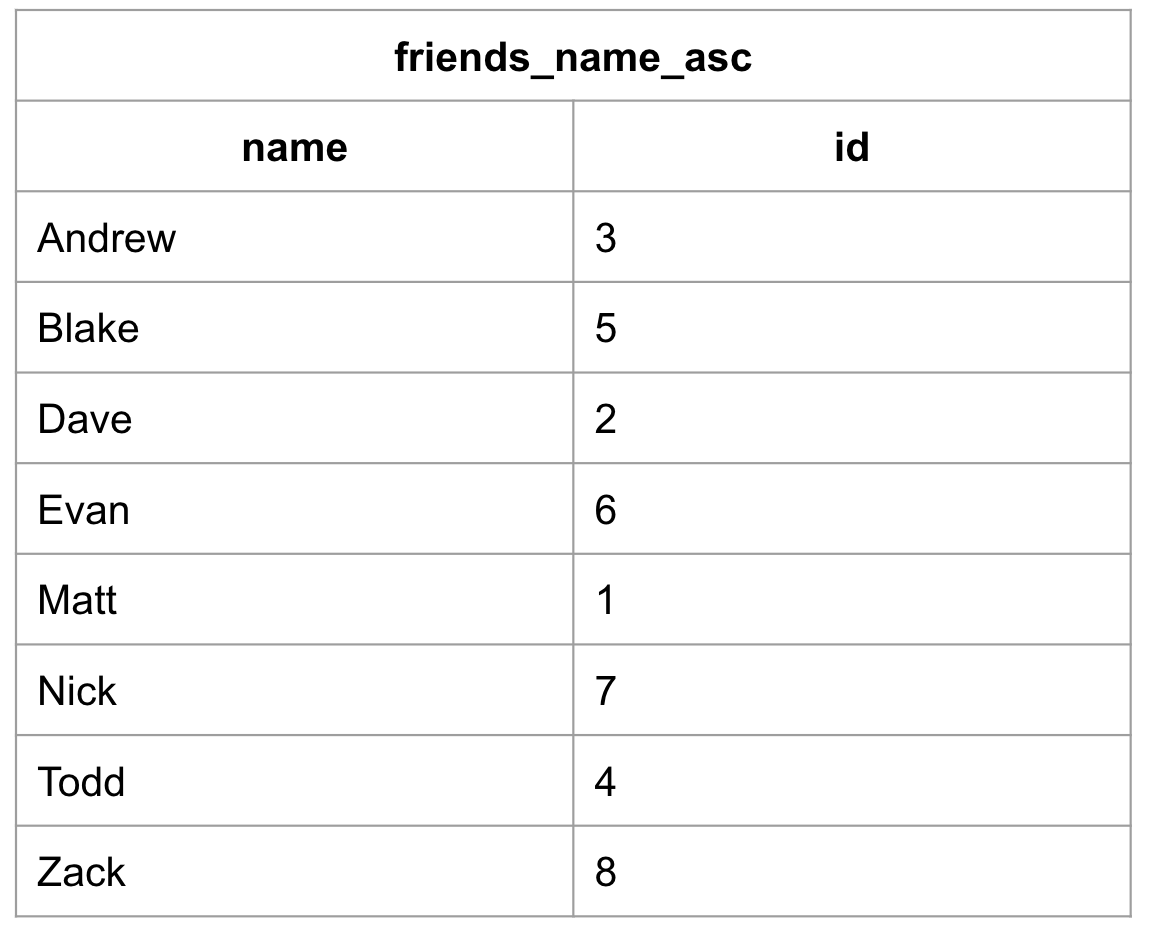
You can create many non-clustered indexes. As of 2008, you can have up to 999 non-clustered indexes in SQL Server and there is no limit in PostgreSQL.

#### **Creating Non-Clustered Databases(PostgreSQL)**

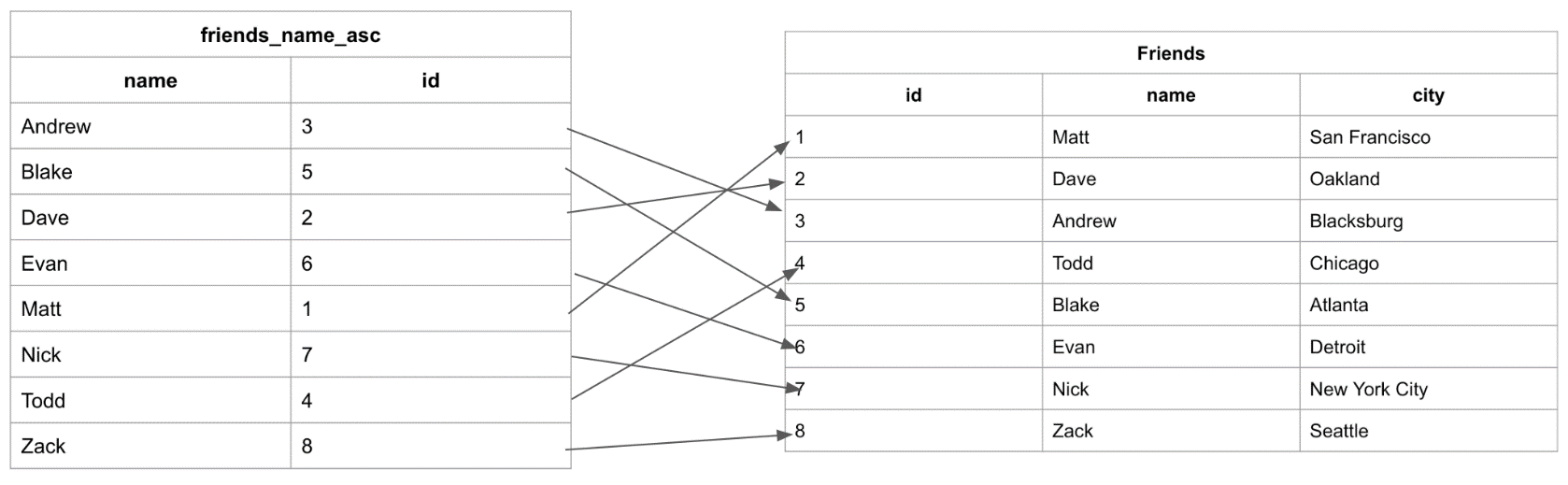
To create an index to sort our friends’ names alphabetically:

**CREATE** **INDEX** friends\_name\_asc **ON** friends(name **ASC**);

This would create an index called “friends\_name\_asc”, indicating that this index is storing the **names** from “friends” stored alphabetically in **ascending** order.



Note that the “city” column is not present in this index. That is because indexes do not store all of the information from the original table. The “id” column would be a pointer back to the original table. The pointer logic would look like this:

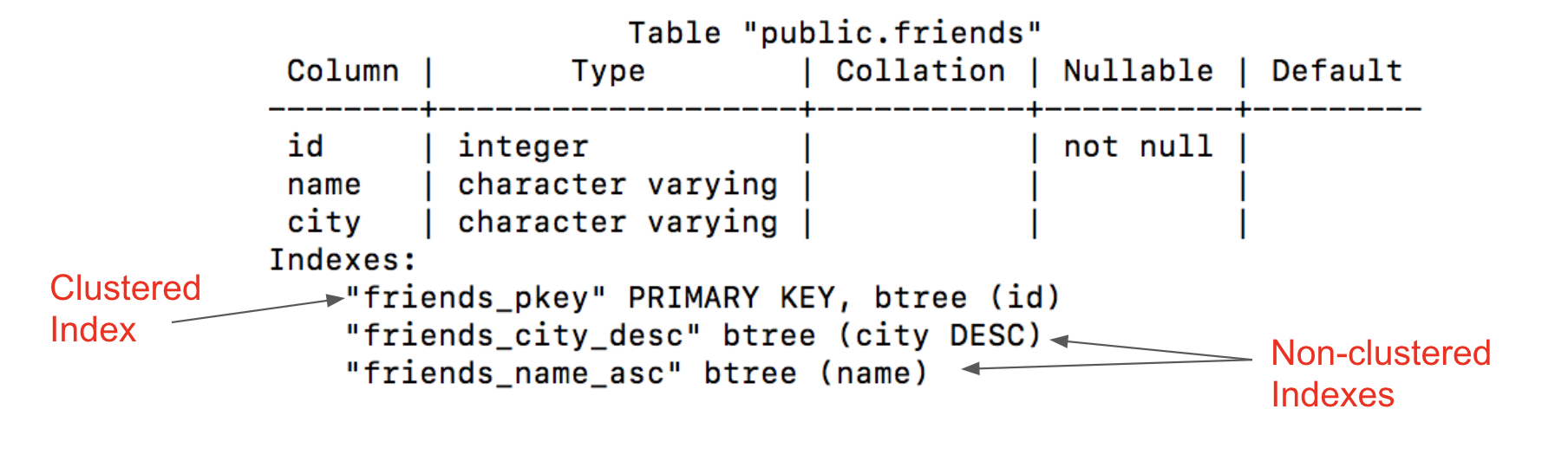


##### **Creating Indexes**

In PostgreSQL, the “\d” command is used to list details on a table, including table name, the table columns and their data types, indexes, and constraints.

The details of our friends table now look like this:

**Query providing details on the friends table**: \d friends;



Looking at the above image, the “friends\_name\_asc” is now an associated index of the “friends” table. That means the [query plan](https://dataschool.com/sql-optimization/what-is-a-query-plan/), the plan that SQL creates when determining the best way to perform a query, will begin to use the index when queries are being made. Notice that “friends\_pkey” is listed as an index even though we never declared that as an index. That is the **clustered index** that was referenced earlier in the article that is automatically created based off of the primary key.

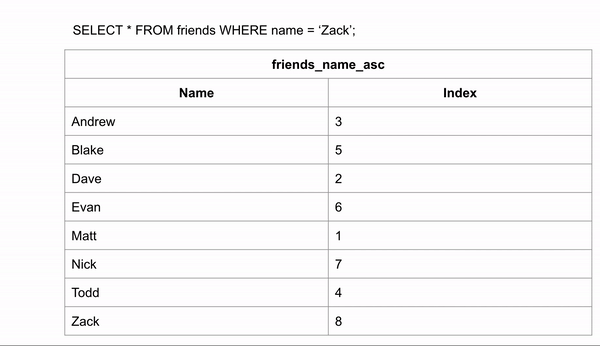
We can also see there is a “friends\_city\_desc” index. That index was created similarly to the names index:

**CREATE** **INDEX** friends\_city\_desc **ON** friends(city **DESC**);

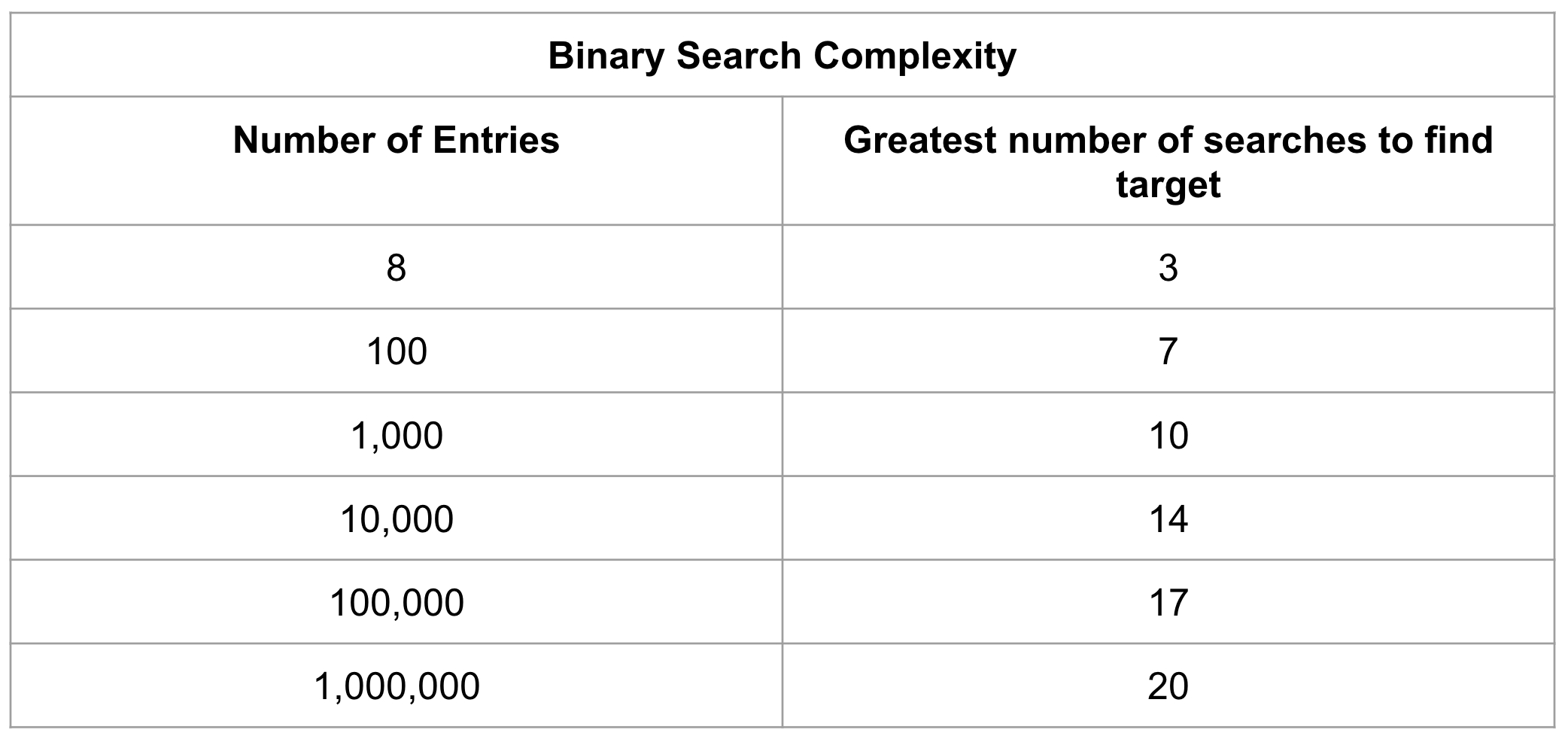
This new index will be used to sort the cities and will be stored in reverse alphabetical order because the keyword “DESC” was passed, short for “descending”. This provides a way for our database to swiftly query city names.

##### **Searching Indexes**

After your non-clustered indexes are created you can begin querying with them. Indexes use an optimal search method known as [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm). Binary searches work by constantly cutting the data in half and checking if the entry you are searching for comes before or after the entry in the middle of the current portion of data. This works well with B-trees because they are designed to start at the middle entry; to search for the entries within the tree you know the entries down the left path will be smaller or before the current entry and the entries to the right will be larger or after the current entry. In a table this would look like:



Comparing this method to the query of the non-indexed table at the beginning of the article, we are able to reduce the total number of searches from eight to three. Using this method, a search of 1,000,000 entries can be reduced down to just 20 jumps in a binary search.



##### **When to use Indexes**

Indexes are meant to speed up the performance of a database, so use indexing whenever it significantly improves the performance of your database. As your database becomes larger and larger, the more likely you are to see benefits from indexing.

##### **When not to use Indexes**

When data is written to the database, the original table (the clustered index) is updated first and then all of the indexes off of that table are updated. Every time a write is made to the database, the indexes are unusable until they have updated. If the database is constantly receiving writes then the indexes will never be usable. This is why indexes are typically applied to databases in data warehouses that get new data updated on a scheduled basis(off-peak hours) and not production databases which might be receiving new writes all the time.

NOTE: The [newest version of Postgres (that is currently in beta](https://www.postgresql.org/about/news/1943/)) will allow you to query the database while the indexes are being updated.

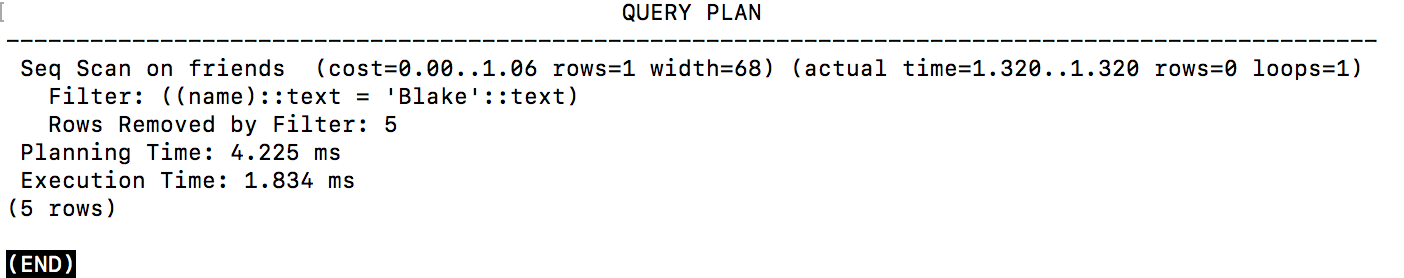
##### **Testing Index performance**

To test if indexes will begin to decrease query times, you can run a set of queries on your database, record the time it takes those queries to finish, and then begin creating indexes and rerunning your tests.

To do this, try using the EXPLAIN ANALYZE clause in PostgreSQL.:

**EXPLAIN** **ANALYZE** **SELECT** **\*** **FROM** friends **WHERE** name **=** 'Blake';

Which on my small database yielded:



This output will tell you which method of search from the query plan was chosen and how long the planning and execution of the query took.

Only create one index at a time because not all indexes will decrease query time.

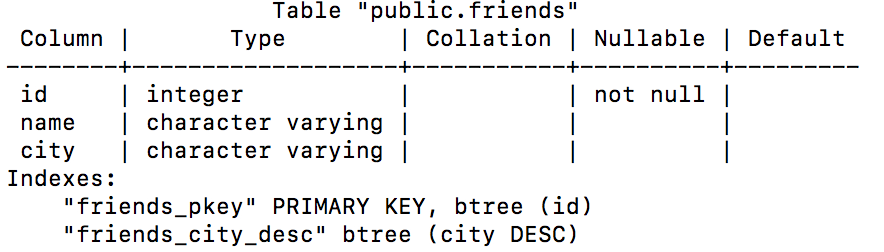
* PostgreSQL’s query planning is pretty efficient, so adding a new index may not affect how fast queries are performed.
* Adding an index will always mean storing more data
* Adding an index will increase how long it takes your database to fully update after a write operation.

If adding an index does not decrease query time, you can simply remove it from the database.

To remove an index use the DROP INDEX command:

**DROP** **INDEX** friends\_name\_asc;

The outline of the database now looks like:



Which shows the successful removal of the index for searching names.

## **Summary**

* Indexing can vastly reduce the time of queries
* Every table with a primary key has one clustered index
* Every table can have many non-clustered indexes to aid in querying
* Non-clustered indexes hold pointers back to the main table
* Not every database will benefit from indexing
* Not every index will increase the query speed for the database