I'm not a robot

A field with a NULL value is a field with no value.If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value. Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation! How to Test for NULL Values?It is not possible to test for NULL values with comparison operators, such as =, mode JSONQuery output mode is JSONsql-> SELECT id, [connections[3:]] AS weakConnections FROM Persons;{"id":3,"weakConnections":[]}{"id":4,"weakConnections":[2]}{"id":2,"weakConnections":[]} {"id":5,"weakConnections":[3]}{"id":1,"weakConnections":[]}5 rows returned Now we run the same query, but without the array constructor. Notice how single items are not contained in an array, and for rows with no match, NULL is returned instead of an empty array. sql-> SELECT id, connections[3:] AS weakConnections FROM Persons; {"id":2,"weakConnections":null}{"id":3,"weakConnections":null}{"id":4,"weakConnections":2}{"id":5,"weakConnections":3}{"id":1,"weakConnections":null} 5 rows returnedsql-> mode COLUMNQuery output mode is COLUMNsql-> As a last example of slice steps, the following query returns the last 3 connections of each person. In this query, the slice step is [size($)-3:]. In this expression, the $ is an implicitly declared variable that references the array that the slice step is applied to. In this example, $ references the connections array. The size() built-in function returns the size (number of elements) of the input array. So, in this example, size($) is the size of the current connections array. Finally, size($)-3 computes the third position from the end of the current connections array. sql-> SELECT id, [connections[size($)-3:]]AS weakConnections FROM Persons; +----+-------------------+ | id | weakConnections | +----+-------------------+ | 5 | 4 | | | 1 | | | 3 | +----+-------------------+ | 4 | 5 | | | 1 | | | 2 | +----+-------------------+ | 3 | 1 | | | 4 | | | 2 | +----+-------------------+ | 2 | 1 | | | 3 | +----+-------------------+ | 1 | 2 | | | 3 | +----+-------------------+5 rows returned We now turn our attention to filter steps on arrays. Like slice steps, filter steps also use the square brackets ([]) syntax. However, what goes inside the [] is different. With filter steps there is either nothing inside the [] or a single expression that acts as a condition (returns a boolean result). In the former case, all the elements of the array are selected (the array is "unnested"). In the latter case, the condition is applied to each element in turn, and if the result is true, the element is selected, otherwise it is skipped. For example: The following query returns the id and connections of persons who are connected to person 4: sql-> SELECT id, connectionsFROM Persons p WHERE p.connections[] =any 4; +----+-------------+ | id | connections | +----+-------------+ | 3 | 1 | | | 4 | | | 2 | +----+-------------+ | 5 | 2 | | | 4 | | | 1 | | | 3 | +----+-------------+2 rows returned In the above query, the expression p.connections[] returns all the connections of a person. Then, the =any operator returns true if this sequence of connections contains the number 4. The following query returns the id and connections of persons who are connected with any person having an id greater than 4: sql-> SELECT id, connections FROM Persons pWHERE p.connections[] >any 4; +----+-------------+ | id | connections | +----+-------------+ | 4 | 3 | | | 5 | | | 1 | | | 2 | +----+-------------+1 row returned The following query returns, for each person, the person's last name and the phone numbers with area code 339: sql-> SELECT lastname,[p.address.phones[$element.areacode = 339].number]AS phoneNumbers FROM Persons p; +----------+--------------+ | lastname | phoneNumbers | +----------+--------------+ | Scully | 3414578 | +----------+--------------+ | Smith | 4120211 | | | 8694021 | | | 1205678 | +----------+--------------+ | Morgan | | +----------+--------------+ | Anderson | 1684972 | +----------+--------------+ | Morrison | | +----------+--------------+5 rows returned In the above query, the filter step [$element.areacode = 339] is applied to the phones array of each person. The filter step evaluates the condition $element.areacode = 339 on each element of the array. This condition expression uses the implicitly declared variable $element, which references the current element of the array. An empty array is returned for persons that do not have any phone number in the 339 area code. If we wanted to filter out such persons from the result, we would write the following query: sql-> SELECT lastname,[p.address.phones[$element.areacode = 339].number]AS phoneNumbers FROM Persons p WHERE p.address.phones.areacode =any 339; +----------+--------------+ | lastname | phoneNumbers | +----------+--------------+ | Scully | 3414578 | +----------+--------------+ | Smith | 4120211 | | | 8694021 | | | 1205678 | +----------+--------------+ | Anderson | 1684972 | +----------+--------------+3 rows returned The previous query contains the path expression p.address.phones.areacode. In that expression, the field step .areacode is applied to an array field (phones). In this case, the field step is applied to each element of the array in turn. In fact, the path expression is equivalent to p.address.phones[].areacode. In addition to the implicitly-declared $ and $element variables, the condition inside a filter step can also use the $pos variable (also implicitly declared). $pos references the position within the array of the current element (the element on which the condition is applied). For example, the following query selects the "interesting" connections of each person, where a connection is considered interesting if it is among the 3 strongest connections and connects to a person with an id greater or equal to 4. sql-> SELECT id, [p.connections[$element >= 4 and $pos < 3]]AS interestingConnections FROM Persons p; +----+------------------------+ | id | interestingConnections | +----+------------------------+ | 5 | 4 | +----+------------------------+ | 4 | 5 | +----+------------------------+ | 3 | 4 | +----+------------------------+ | 2 | | +----+------------------------+ | 1 | | +----+------------------------+5 rows returned Finally, two arrays can be compared with each other using the usual comparison operators (=, !=, >, >=, >, and >=). For example the following query constructs the array [1,3] and selects persons whose connections array is equal to [1,3]. sql-> SELECT lastname FROM Persons pWHERE p.connections = [1,3]; +----------+ | lastname | +----------+ | Anderson | +----------+1 row returned Page 7 You can use a field step to select the value of a field from a record. For example, to return the id, last name, and city of persons who reside in Florida: sql-> SELECT id, lastname, p.address.cityFROM Persons p WHERE p.address.state = "FL"; +----+----------+------------+ | id | lastname | city | +----+----------+------------+ | 3 | Morgan | Middleburg | +----+----------+------------+1 row returned In the above query, the path expression p.address.state consists of 2 field steps: .address selects the address field of the current row (rows can be viewed as records, whose fields are the row columns), and .state selects the state field of the current address. The example record contains an array of phone numbers. You can form queries against that array using a combination of path steps and sequence comparison operators. For example, to return the last name of persons who have a phone number with area code 423: sql-> SELECT lastname FROM Personsp WHERE p.address.phones.areacode =any 423; +----------+ | lastname | +----------+ | Morrison | +----------+1 row returned In the above query, the path expression p.address.phones.areacode returns all the area codes of a person. Then, the =any operator returns true if this sequence of area codes contains the number 423. Notice also that the field step .areacode is applied to an array field (phones). This is allowed if the array contains records or maps. In this case, the field step is applied to each element of the array in turn. The following example returns all the persons who had three connections. Notice the use of [] after connections: it is an array filter step, which returns all the elements of the connections array as a sequence (it is unnesting the array). sql-> SELECT id, firstName, lastName, connections from Persons where connections[] =any 3 ORDER BY id; +----+-----------+----------+-------------+ | id | firstName | lastName | connections | +----+-----------+----------+-------------+ | 1 | David | Morrison | 2 | | | | | 3 | +----+-----------+----------+-------------+ | 2 | John | Anderson | 1 | | | | | 3 | +----+-----------+----------+-------------+ | 4 | Peter | Smith | 3 | | | | | 5 | | | | | 1 | | | | | 2 | +----+-----------+----------+-------------+ | 5 | Dana | Scully | 2 | | | | | 4 | | | | | 1 | | | | | 3 | +----+-----------+----------+-------------+4 rows returned This query can use ORDER BY to sort the results because the sort is being performed on the table's primary key. The next section shows sorting on non-primary key fields through the use of indexes. For more examples of querying against data contained in arrays, see Working With Arrays. Page 8 To sort the results from a SELECT statement using a field that is not the table's primary key, you must first create an index for the column of choice. For example, for the next table, to query based on a Timestamp and sort the results in descending order by the timestamp, create an index: sql-> SELECT id, firstname, lastname, lastLogin FROM Persons; +----+-----------+----------+-------------------------+ | id | firstname | lastname | lastLogin | +----+-----------+----------+-------------------------+ | 3 | John | Morgan | 2016-11-29T08:21:35.4971 | | 4 | Peter | Smith | 2016-10-19T09:18:05.5555 | | 2 | John | Anderson | 2016-11-28T13:01:11.2088 | | 5 | Dana | Scully | 2016-11-08T09:16:46.3929 | | 1 | David | Morrison | 2016-10-29T18:43:59.8319 | +----+-----------+----------+-------------------------+ 5 rows returned sql-> CREATE INDEX tsidx1 on Persons (lastLogin);Statement completed successfullysql-> SELECT id, firstname, lastname, lastLogin FROM Persons ORDER BY lastLogin DESC; +----+-----------+----------+-------------------------+ | id | firstname | lastname | lastLogin | +----+-----------+----------+-------------------------+ | 3 | John | Morgan | 2016-11-29T08:21:35.4971 | | 2 | John | Anderson | 2016-11-28T13:01:11.2088 | | 5 | Dana | Scully | 2016-11-08T09:16:46.3929 | | 1 | David | Morrison | 2016-10-29T18:43:59.8319 | | 4 | Peter | Smith | 2016-10-19T09:18:05.5555 | +----+-----------+----------+-------------------------+5 rows returned SQL for Oracle NoSQL Database can also sort query results by the values of nested records. To do so, create an index of the nested field (or fields). For example, you can create an index of address.state from the Persons table, and then order by state: sql-> CREATE INDEX indx1 on Persons (address.state);Statement completed successfullysql-> SELECT id, $p.address.state FROMPersons $p ORDER BY $p.address.state; +----+-------+ | id | state | +----+-------+ | 3 | FL | | 4 | MA | | 5 | NJ | | 1 | TN | | 2 | WI | +----+-------+5 rows returned To learn more about indexes, see Working With Indexes. Page 9 The path steps applicable to maps are field and filter steps. Slice steps do not make sense for maps, because maps are unordered, and as a result, their entries do not have any fixed positions. You can use a field step to select the value of a field from a map. For example, to return the lastname and the food expenses of all persons: sql-> SELECT lastname, p.expenses.foodFROM Persons p; +----------+------+ | lastname | food | +----------+------+ | Morgan | 2000 | | Morrison | 1000 | | Scully | 900 | | Smith | 6000 | | Anderson | 1700 | +----------+------+5 rows returned In the above query, the path expression p.expenses.food consists of 2 field steps: .expenses selects the expenses field of the current row and .food selects the value of the food field/entry from the current expenses map. To return the lastname and amount spent on travel for each person who spent less than $3000 on food: sql-> SELECT lastname, p.expenses.travelFROM Persons p WHERE p.expenses.food < 3000; +----------+--------+ | lastname | travel | +----------+--------+ | Scully | NULL | | Morgan | 700 | | Anderson | 2100 | | Morrison | NULL | +----------+--------+4 rows returned Notice that NULL is returned for persons who did not have any travel expenses. Filter steps are performed using either the .values() or .keys() path steps. To select values of map entries, use .values(). To select keys of map entries, use .keys(). If no condition is used in these steps, all the values or keys of the input map are selected. If the steps do contain a condition expression, the condition is evaluated for each entry, and the value or key of the entry is selected/skipped if the result is true/false. The implicitly-declared variables $key and $value can be used inside a map filter condition. $key references the key of the current entry and $value references the associated value. Notice that, contrary to arrays, the $pos variable can not be be used inside map filters (because map entries do not have fixed positions). To show, for each user, their id and the expense categories where they spent more than $1000: sql-> SELECT id, p.expenses.keys($value > 1000) as Expenses from Persons p; +----+--------------------+ | id | Expenses | +----+--------------------+ | 4 | clothes | | | food | | | shoes | +----+--------------------+ | 3 | food | +----+--------------------+ | 2 | food | | | travel | +----+--------------------+ | 5 | clothes | +----+--------------------+ | 1 | NULL | +----+--------------------+ To return the id and the expense categories in which the user spent more than they spent on clothes, use the following filter step expression. In this query, the context-item variable ($) appearing in the filter step expression [$value > $.clothes] refers to the expenses map as a whole. sql-> SELECT id, p.expenses.keys($value > $.clothes) FROM Persons p; +----+--------------------+ | id | Column_2 | +----+--------------------+ | 3 | NULL | +----+--------------------+ | 2 | NULL | +----+--------------------+ | 5 | NULL | +----+--------------------+ | 1 | NULL | +----+--------------------+ | 4 | food | +----+--------------------+ To return the id and expenses data of any person who spent more on any category than what they spent on food: sql-> SELECT id, p.expensesFROM Persons pWHERE p.expenses.values() >any p.expenses.food; +----+--------------------+ | id | expenses | +----+--------------------+ | 5 | clothes | 1500 | | | food | 900 | | | shoes | 1000 | +----+--------------------+ | 2 | books | 100 | | | food | 1700 | | | travel | 2100 | +----+--------------------+2 rows returned To return the id of all persons who consumed more than $2000 in any category other than food: sql-> SELECT id FROM Persons pWHERE p.expenses.values($key != "food") >any 2000; +----+ | id | +----+ | 2 | +----+1 row returned Page 10 The size function can be used to return the size (number of fields/entries) of a complex item (record, array, or map). For example: To return the id and the number of phones that each person has: sql-> SELECT id, size(p.address.phones)AS registeredphones FROM Persons p; +----+------------------+ | id | registeredphones | +----+------------------+ | 5 | 3 | | 3 | 2 | | 4 | 4 | | 2 | 1 | | 1 | 1 | +----+------------------+5 rows returned To return the id and the number of expenses categories for each person: has: sql-> SELECT id, size(p.expenses) AScategories FROM Persons p; +----+------------+ | id | categories | +----+------------+ | 4 | 4 | | 3 | 3 | | 2 | 3 | | 1 | 2 | | 5 | 3 | +----+------------+5 rows returned To return for each person their id and the number of expenses categories for which the expenses were more than 2000: sql-> SELECT id, size([p.expenses.values($value > 2000)]) AS expensiveCategories FROM Persons p; +----+---------------------+ | id | expensiveCategories | +----+---------------------+ | 3 | 0 | | 2 | 1 | | 5 | 0 | | 1 | 0 | | 4 | 1 | +----+---------------------+5 rows returned Page 11 This chapter provides examples on working with JSON data. If you want to follow along with the examples, get the Examples download from here and run the SQLJSONExamples script found in the sqlfolder. This creates the table and imports the data used. JSON data is written to JSON data columns by providing a JSON object. This object can contain any valid JSON data. The input data is parsed and stored internally as Oracle NoSQL Database datatypes: When numbers are encountered, they are converted to integer, long, or double items, depending on the actual value of the number (float items are not used for JSON). Strings in the input text are mapped to string items. Boolean values are mapped to boolean items. JSON nulls are mapped to JSON null items. When an array is encountered in the input text, an array item is created whose type is Array(JSON). This is done unconditionally, no matter what the actual contents of the array might be. When a JSON object is encountered in the input text, a map item is created whose type is Map(JSON), unconditionally. The remainder of this chapter provides an overview to querying JSON data. Page 12 The SQLJSONExample is available to illustrate JSON usage. This script creates the following table: create table if not exists JSONPersons (id integer, person JSON, primary key (id)); The script imports the following table rows. Notice that the content for the person column, which is of type JSON contains a JSON object. That object contains a series of fields which represent our person. We have deliberately included inconsistent information in this example so as to illustrate how to handle various queries when working with JSON data. { "id":1, "person" : { "firstname":"David", "lastname":"Morrison", "age":25, "income":100000, "lastLogin" : "2016-10-29T18:43:59.8319", "address":{"street":"150 Route 2", "city":"Antioch", "state":"TN", "zipcode" : 37013, "phones":[{"type":"home", "areacode":423, "number":8634379}] }, "connections":[2, 3], "expenses":{"food":1000, "gas":180} }}{ "id":2, "person" : { "firstname":"John", "lastname":"Anderson", "age":35, "income":100000, "lastLogin" : "2016-11-28T13:01:11.2088", "address":{"street":"187 Hill Street", "city":"Beloit", "state":"WI", "zipcode" : 53511, "phones":[{"type":"home", "areacode":339, "number":1684972}] }, "connections":[1, 3], "expenses":{"books":100, "food":1700, "travel":2100} }}{ "id":3, "person" : { "firstname":"John", "lastname":"Morgan", "age":38, "income":100000000, "lastLogin" : "2016-11-29T08:21:35.4971", "address":{"street":"187 Aspen Drive", "city":"Middleburg", "state":"FL", "phones":[{"type":"work", "areacode":305, "number":1234079}, {"type":"home", "areacode":305, "number":2066401}] }, "connections":[1, 4, 2], "expenses":{"food":2000, "travel":700, "gas":10} }}{ "id":4, "person": { "firstname":"Peter", "lastname":"Smith", "age":38, "income":80000, "lastLogin" : "2016-10-19T09:18:05.5555", "address":{"street":"364 Mulberry Street", "city":"Leominster", "state":"MA", "phones":[{"type":"work", "areacode":339, "number":4120211}, {"type":"work", "areacode":339, "number":8694021}, {"type":"home", "areacode":339, "number":1205678}, null, {"type":"home", "areacode":305, "number":8064321}] }, "connections":[3, 5, 1, 2], "expenses":{"food":6000, "books":240, "clothes":2000, "shoes":1200} }}{ "id":5, "person" : { "firstname":"Dana", "lastname":"Scully", "age":47, "income":400000, "lastLogin" : "2016-11-08T09:16:46.3929", "address":{"street":"427 Linden Avenue", "city":"Monroe Township", "state":"NJ", "phones":[{"type":"work", "areacode":201, "number":3213267}, {"type":"work", "areacode":201, "number":8765421}, {"type":"home", "areacode":339, "number":3414578}] }, "connections":[2, 4, 1, 3], "expenses":{"food":900, "shoes":1000, "clothes":1500} }}{ "id":6, "person" : { "mynumber":5, "myarray":[1,2,3,4] }}{ "id":7, "person" : { "mynumber":"5", "myarray":["1","2","3","4"] }} You run the SQLJSONExamples script using the load command: > cd /examples/sql> java -jar /lib/sql.jar -helper-hosts : \-store load \-file /examples/sql/SQLJSONExamples.cliPage 13 Because JSON is parsed and stored internally in native data formats with Oracle NoSQL Database, querying JSON data is no different than querying data in other column types. See Simple SELECT Queries and Working with complex data for introductory examples of how to form these queries. In our JSONPersons example, all of the data for each person is contained in a column of type JSON called person. This data is presented as a JSON object, and mapped internally into a Map(JSON) type. You can query information in this column as you would query a Map of any other type. For example: sql-> SELECT id, j.person.lastname, j.person.age FROM JSONPersons j; +----+--------------------+------------+ | id | lastname | age | +----+--------------------+------------+ | 3 | Morgan | 38 | +----+--------------------+------------+ | 2 | Anderson | 35 | +----+--------------------+------------+ | 5 | Scully | 47 | +----+--------------------+------------+ | 1 | Morrison | 25 | +----+--------------------+------------+ | 4 | Smith | 38 | +----+--------------------+------------+ | 6 | NULL | NULL | +----+--------------------+------------+ | 7 | NULL | NULL | +----+--------------------+------------+7 rows returned The last two rows in returned from this query contain all NULLs. This is because those rows were populated using JSON objects that are different than the objects used to populate the rest of the table. This capability of JSON is both a strength and a weakness. As a plus, you can modify your schema easily. However, if you are not careful, you can end up with tables containing dissimilar data in both large and small ways. Because the JSON object is stored as a map, you can use normal map step functions on the column. For example: sql-> SELECT id, j.person.expenses.keys($value > 1000) as Expenses from JSONPersons j;+----+--------------------+ | id | Expenses | +----+--------------------+ | 3 | food | +----+--------------------+ | 2 | food | | | travel | +----+--------------------+ | 4 | clothes | | | food | | | shoes | +----+--------------------+ | 6 | NULL | +----+--------------------+ | 5 | clothes | +----+--------------------+ | 7 | NULL | +----+--------------------+ | 1 | NULL | +----+--------------------+7 rows returned Here, id 1 is NULL because that user had no expenses greater than $1000, while id 6 and 7 are NULL because they have no j.person.expenses field. Page 14 As we saw in the previous section, different rows in the same table can have dissimilar information in them when a column type is JSON. To identify whether desired information exists for a given JSON column, use the EXISTS operator. For example, some of the JSON persons have a zip code entered for their address, and others do not. Use this query to see all the users with a zipcode: sql-> SELECT id, j.person.address AS Address FROM JSONPersons j WHERE EXISTS j.person.address.zipcode; +----+-------------------------------+ | id | Address | +----+-------------------------------+ | 2 | city | Beloit | | | phones | | | areacode | 339 | | | number | 1684972 | | | type | home | | | state | WI | | | street | 187 Hill Street | | | zipcode | 53511 | +----+-------------------------------+ | 1 | city | Antioch | | | phones | | | areacode | 423 | | | number | 8634379 | | | type | home | | | state | TN | | | street | 150 Route 2 | | | zipcode | 37013 | +----+-------------------------------+2 rows returned When querying data for inconsistencies, it is often more useful to see all rows where information is missing by using WHERE NOT EXISTS: sql-> SELECT * FROM JSONPersons j WHERE NOT EXISTS j.person.lastname; +----+------------------+ | id | person | +----+------------------+ | 7 | myarray | | | 1 | | | 2 | | | 3 | | | 4 | | | mynumber | 5 | +----+------------------+ | 6 | myarray | | | 1 | | | 2 | | | 3 | | | 4 | | | mynumber | 5 | +----+------------------+1 row returned Page 15 All arrays found in a JSON input stream are stored internally as ARRAY(JSON). This means that it is possible for the array to have inconsistent types for its members. In our example, the phones array for user id 4 contains a null element: sql-> SELECT j.person.address.phones FROM JSONPersons j WHERE j.id=4; +--------------------+ | phones | +--------------------+ | areacode | 339 | | number | 4120211 | | type | work | | | | areacode | 339 | | number | 8694021 | | type | work | | | | areacode | 339 | | number | 1205678 | | type | home | | null | | | | areacode | 305 | | number | 8064321 | | type | home | +--------------------+ A way to discover this in your table is to examine the phones array for null values: sql-> SELECT id, j.person.address.phones FROM JSONPersons j WHERE j.person.address.phones[] =any null; +----+--------------------+ | id | phones | +----+------------------+ | 4 | areacode | 339 | | | number | 4120211 | | | type | work | | | | | areacode | 339 | | | number | 8694021 | | | type | work | | | | | areacode | 339 | | | number | 1205678 | | | type | home | | | null | | | | | areacode | 305 | | | number | 8064321 | | | type | home | +----+--------------------+1 row returned Notice the use of the array filter step ([]) in the previous query. This is needed to unpack the array into a sequence so that the =any comparison operator can be used with it. Page 16 The example data contains a couple of rows with unusual data: { "id":6, "person" : { "mynumber":5, "myarray":[1,2,3,4] }}{ "id":7, "person" : { "mynumber":"5", "myarray":["1","2","3","4"] }} You can locate them using the query: sql-> SELECT * FROM JSONPersons j WHERE EXISTS j.person.mynumber; +----+------------------+ | id | person | +----+------------------+ | 6 | myarray | | | 1 | | | 2 | | | 3 | | | 4 | | | mynumber | 5 | +----+------------------+ | 7 | myarray | | | 1 | | | 2 | | | 3 | | | 4 | | | mynumber | 5 | +----+------------------+2 rows returned However, notice that these two rows actually contain numbers stored as different types. ID 6 stores integers while ID 7 stores strings. You can select a row based on its type: sql-> SELECT * FROM JSONPersons j WHERE j.person.mynumber IS OF TYPE (integer); +----+------------------+ | id | person | +----+------------------+ | 6 | myarray | | | 1 | | | 2 | | | 3 | | | 4 | | | mynumber | 5 | +----+------------------+ Notice that if you use IS NOT OF TYPE then every row in the table is returned except id 6. This is because for all the other rows, j.person.mynumber evaluates to jnull, which is not an integer. sql-> SELECT id FROM JSONPersons j WHERE j.person.mynumber IS NOT OF TYPE (integer); +----+ | id | +----+ | 3 | | 2 | | 5 | | 4 | | 1 | | 7 | +----+6 rows returned To solve this problem, also check for the existence of j.person.mynumber: sql-> SELECT id from JSONPersons j WHERE EXISTS j.person.mynumber and j.person.mynumber IS NOT OF TYPE (integer); +----+ | id | +----+ | 7 | +----+1 row returned You can also perform type checking based on the type of data contained in the array. Recall that our rows contain arrays with integers and arrays with strings. You can return the row with just the array of strings using: sql-> SELECT id, j.person.myarray FROM JSONPersons j WHERE j.person.myarray[] IS OF TYPE (string+); +----+-------------------+ | id | myarray | +----+-------------------+ | 7 | 1 | | | 2 | | | 3 | | | 4 | +----+-------------------+1 row returned Here, we use the array filter step ([]) in the WHERE clause to unpack the array into a sequence. This allows is-of-type to iterate over the sequence, checking the type of each element. If every element in the sequence matches the identified type (string, in this case), then the is-of-type returns true. Also notice that the query uses the + cardinality modifier. This means that is-of-type will return true only if the input sequence (myarray[], in this case) contains ONE OR MORE elements that match the identified type (string). If we used *, then 0 or more elements would have to match the identified type in order for true to return. Because our table contains a mix of rows with different schema, the result is that every row except id 6 is returned: sql-> SELECT id, j.person.myarray FROM JSONPersons j WHERE j.person.myarray[] IS OF TYPE (string*); +----+-------------------+ | id | myarray | +----+-------------------+ | 3 | NULL | +----+-------------------+ | 5 | NULL | +----+-------------------+ | 1 | NULL | +----+-------------------+ | 7 | 1 | | | 2 | | | 3 | | | 4 | +----+------------------+ | 4 | NULL | +----+-------------------+ | 2 | NULL | +----+-------------------+6 rows returned Finally, if we do not provide a cardinality modifier at all, then is-of-type returns true if ONE AND ONLY one member of the input sequence matches the identified type. In this example, the result is that no rows are returned. sql-> SELECT id, j.person.myarray FROM JSONPersons j WHERE j.person.myarray[] IS OF TYPE (string);0 row returned Page 17 On import, Oracle NoSQL Database stores JSON objects as MAP(JSON). This means you can use map filter steps with your JSON objects. For example, if you want to visually examine the JSON fields in use by your rows: sql-> SELECT id, j.person.keys() FROM JSONPersons j; +----+------------------------+ | id | Column_2 | +----+------------------------+ | 4 | address | | | age | | | connections | | | expenses | | | firstname | | | income | | | lastLogin | | | lastname | +----+------------------------+ | 6 | myarray | | | mynumber | +----+------------------------+ | 3 | address | | | age | | | connections | | | expenses | | | firstname | | | income | | | lastLogin | | | lastname | +----+------------------------+ | 5 | address | | | age | | | connections | | | expenses | | | firstname | | | income | | | lastLogin | | | lastname | +----+------------------------+ | 1 | address | | | age | | | connections | | | expenses | | | firstname | | | income | | | lastLogin | | | lastname | +----+------------------------+ | 7 | myarray | | | mynumber | +----+--------------------+ | 2 | address | | | age | | | connections | | | expenses | | | firstname | | | income | | | lastLogin | | | lastname | +----+------------------------+7 rows returned Page 18 You can cast one data type to another using the cast expression. In JSON, casting is particularly useful for timestamp information because JSON has no equivalent to the Oracle NoSQL Database Timestamp data type. Instead, the timestamp information is carried in a JSON object as a string. To work with it as a Timestamp, use cast. In Working with Timestamps we showed how to work with the timestamp data type. In this case, what you do is no different except you must cast both sides of the expression. Also, because the left side of the expression is a sequence, you must specify a type quantifier (* in this case): sql-> SELECT id, j.person.firstname, j.person.lastname, j.person.lastLogin FROM JSONPersons j WHERE CAST(j.person.lastLogin AS TIMESTAMP*) > CAST("2016-11-01" AS TIMESTAMP) AND CAST(j.person.lastLogin AS TIMESTAMP*) < CAST("2016-11-30" AS TIMESTAMP); +----+-----------+-------------+--------------------------+ | id | firstname | lastname | lastLogin | +----+-----------+-------------+--------------------------+ | 3 | John | Morgan | 2016-11-29T08:21:35.4971 | +----+-----------+-------------+--------------------------+ | 2 | John | Anderson | 2016-11-28T13:01:11.2088 | +----+-----------+-------------+--------------------------+ | 5 | Dana | Scully | 2016-11-08T09:16:46.3929 | +----+-----------+-------------+--------------------------+3 rows returned As another example, you can cast to an integer and then operate on that number: sql-> SELECT id, j.person.mynumber, CAST(j.person.mynumber as integer) * 10 AS TenTimes FROM JSONPersons j WHERE EXISTS j.person.mynumber; +----+--------------------+----------+ | id | mynumber | TenTimes | +----+--------------------+----------+ | 7 | 5 | 50 | +----+--------------------+----------+ | 6 | 5 | 50 | +----+--------------------+----------+ If you want to operate on just the row that contains the number as a string, use IS OF TYPE: sql-> SELECT id, j.person.mynumber, CAST(j.person.mynumber as integer) * 10 AS TenTimes FROM JSONPersons j WHERE EXISTS j.person.mynumber AND j.person.mynumber IS OF TYPE (string); +----+--------------------+----------+ | id | mynumber | TenTimes | +----+--------------------+----------+ | 7 | 5 | 50 | +----+--------------------+----------+ Page 19 A searched case expression can be helpful in identifying specific problems with the JSON data in your JSON columns. The example data we have been using in this chapter sometimes provides a JSONPersons.address field, and sometimes it does not. When an address is present, sometimes it provides a zipcode, and sometimes it does not. We can use a searched case expression to identify and describe the specific problem with each row. sql-> SELECT id, CASE WHEN NOT EXISTS j.person.address THEN j.person.keys() WHEN NOT EXISTS j.person.address.zipcode THEN "No Zipcode" ELSE j.person.address.zipcodeENDFROM JSONPersons j; +----+----------------------+ | id | Column_2 | +----+----------------------+ | 4 | No Zipcode | +----+----------------------+ | 3 | No Zipcode | +----+----------------------+ | 5 | No Zipcode | +----+----------------------+ | 1 | 37013 | +----+----------------------+ | 7 | myarray | | | mynumber | +----+----------------------+ | 6 | myarray | | | mynumber | +----+----------------------+ | 2 | 53511 | +----+----------------------+7 rows returned We can improve the report by adding a third column that uses a second searched case expression: sql-> SELECT id,CASE WHEN NOT EXISTS j.person.address THEN "No Address" WHEN NOT EXISTS j.person.address.zipcode THEN "No Zipcode" ELSE j.person.address.zipcodeEND,CASE WHEN NOT EXISTS j.person.address THEN j.person.keys() ELSE j.person.addressENDFROM JSONPersons j; +----+----------------------+-----------------------------------+ | id | Column_2 | Column_3 | +----+----------------------+-----------------------------------+ | 3 | No Zipcode | city | Middleburg | | | | phones | | | | areacode | 305 | | | | number | 1234079 | | | | type | work | | | | | | | | areacode | 305 | | | | number | 2066401 | | | | type | home | | | | state | FL | | | | street | 187 Aspen Drive | +----+----------------------+-----------------------------------+ | 2 | 53511 | city | Beloit | | | | phones | | | | areacode | 339 | | | | number | 1684972 | | | | type | home | | | | state | WI | | | | street | 187 Hill Street | | | | zipcode | 53511 | +----+----------------------+-----------------------------------+ | 5 | No Zipcode | city | Monroe Township | | | | phones | | | | areacode | 201 | | | | number | 3213267 | | | | type | work | | | | | | | | areacode | 201 | | | | number | 8765421 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 3414578 | | | | type | home | | | | state | NJ | | | | street | 427 Linden Avenue | +----+----------------------+-----------------------------------+ | 1 | 37013 | city | Antioch | | | | phones | | | | areacode | 423 | | | | number | 8634379 | | | | type | home | | | | state | TN | | | | street | 150 Route 2 | | | | zipcode | 37013 | +----+----------------------+-----------------------------------+ | 7 | No Address | myarray | | | | mynumber | +----+----------------------+-----------------------------------+ | 4 | No Zipcode | city | Leominster | | | | phones | | | | areacode | 339 | | | | number | 4120211 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 8694021 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 1205678 | | | | type | home | | | | null | | | | | | | | areacode | 305 | | | | number | 8064321 | | | | type | home | | | | state | MA | | | | street | 364 Mulberry Street | +----+----------------------+-----------------------------------+ | 6 | No Address | myarray | | | | mynumber | +----+----------------------+-----------------------------------+7 rows returned Finally, it is possible to nest search case expressions. Our sample data also has a spurious null in the phones array (see id 4). We can report that in the following way (output is modified slightly to fit in the space allowed): sql-> SELECT id,CASE WHEN EXISTS j.person.address THEN CASE WHEN EXISTS j.person.address.zipcode THEN CASE WHEN j.person.address.phones[] =any null THEN "Zipcode exists but null in the phones array" ELSE j.person.address.zipcode END WHEN j.person.address.phones[] =any null THEN "No zipcode and null in phones array" ELSE "No zipcode" END ELSE "No Address"END,CASE WHEN NOT EXISTS j.person.address THEN j.person.keys() ELSE j.person.addressEND FROM JSONPersons j; +----+----------------------+-----------------------------------+ | id | Column_2 | Column_3 | +----+----------------------+-----------------------------------+ | 3 | No zipcode | city | Middleburg | | | | phones | | | | areacode | 305 | | | | number | 1234079 | | | | type | work | | | | | | | | areacode | 305 | | | | number | 2066401 | | | | type | home | | | | state | FL | | | | street | 187 Aspen Drive | +----+----------------------+-----------------------------------+ | 2 | 53511 | city | Beloit | | | | phones | | | | areacode | 339 | | | | number | 1684972 | | | | type | home | | | | state | WI | | | | street | 187 Hill Street | | | | zipcode | 53511 | +----+----------------------+-----------------------------------+ | 5 | No zipcode | city | Monroe Township | | | | phones | | | | areacode | 201 | | | | number | 3213267 | | | | type | work | | | | | | | | areacode | 201 | | | | number | 8765421 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 3414578 | | | | type | home | | | | state | NJ | | | | street | 427 Linden Avenue | +----+----------------------+-----------------------------------+ | 1 | 37013 | city | Antioch | | | | phones | | | | areacode | 423 | | | | number | 8634379 | | | | type | home | | | | state | TN | | | | street | 150 Route 2 | | | | zipcode | 37013 | +----+----------------------+-----------------------------------+ | 7 | No Address | myarray | | | | mynumber | +----+----------------------+-----------------------------------+ | 4 | No zipcode and null | city | Leominster | | | in phones array | phones | | | | areacode | 339 | | | | number | 4120211 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 8694021 | | | | type | work | | | | | | | | areacode | 339 | | | | number | 1205678 | | | | type | home | | | | null | | | | | | | | areacode | 305 | | | | number | 8064321 | | | | type | home | | | | state | MA | | | | street | 364 Mulberry Street | +----+----------------------+---+ | 6 | No Address | myarray | | | | mynumber | +----+----------------------+-----------------------------------+7 rows returned Page 20 The GeoJSON specification (defines the structure and content of JSON objects representing geographical shapes on earth (called geometries). Oracle NoSQL Database implements several functions that interpret JSON geometry objects. The functions also let you search table rows containing geometries that satisfy certain conditions. Search is made efficient through the use of special indexes, as described in the SQL Reference Guide. Page 21 As described, all kinds of geometries are specified in terms of a set of positions. However, for line strings and polygons, the actual geometrical shape is formed by lines connecting their positions. The GeoJSON specification defines a line between two points as the straight line that connects the points in the (flat) cartesian coordinate system, whose horizontal and vertical axes are the longitude and latitude, respectively. More precisely, the coordinates of every point on a line that does not cross the antimeridian between a point P1 = (lon1, lat1) and P2 = (lon2, lat2) can be calculated as: P = (lon, lat) = (lon1 + (lon2 - lon1) * t, lat1 + (lat2 - lat1) * t) with t being a real number, greater than or equal to 0, and less than or equal to 1. Unlike the GeoJSON specification, the Oracle NoSQL Database uses a geodetic coordinate system, as defined in the World Geodetic System, WGS84, (. A geodetic line between two points is the shortest line that can be drawn between the two points on the ellipsoidal surface of the earth. Description of the illustration geodedic1.png Page 22 The GeoJSON specification (states that for a JSON object to be a geometry, it requires two fields, type and coordinates. The value of the type field specifies the kind of geometric shape the object describes. The value of the type field must be one of the following strings, corresponding to different kinds of geometries: Point LineSegment Polygon MultiPoint MultiLineString MultiPolygon GeometryCollection The coordinates value is an array with elements that define the geometrical shape. An exception to this is the GeometryCollection type, which is described below. The coordinates value depends on the geometric shape, but in all cases, specifies a number of positions. A position defines a position on the surface of the earth as an array of two double numbers, where the first number is the longitude and the second number is the latitude. Longitude and latitude are specified as degrees and must range between -180 +180 and -90 +90, respectively. The kinds of geometries are defined as follows, each with an example of such an object: Point For type Point, the coordinates field is a single position:{ "type" : "point", "coordinates" : [23.549, 35.2908] } LineString A LineString is one or more connected lines, with the end-point of one line being the start-point of the next. The coordinates field is an array of two or more positions. The first position is the start point of the first line, and each subsequent position is the end point of the previous line and the start of the next line. Lines can cross each other. { "type" : "LineString","coordinates" : [[-121.9447, 37.2975],[-121.9500, 37.3171],[-121.9892, 37.3182],[-122.1554, 37.3882],[-122.2899, 37.4589],[-122.4273, 37.6032],[-122.4304, 37.6267], [-122.3975, 37.6144]]} Polygon A polygon defines a surface area by specifying its outer perimeter and the perimeters of any potential holes inside the area. More precisely, a polygon consists of one or more linear rings, where (a) a linear ring is a closed LineString with four or more positions, (b) the first and last positions are equivalent, and they must contain identical values, (c) a linear ring is the boundary of a surface or the boundary of a hole in a surface, and (d) a linear ring must follow the right-hand rule with respect to the area it bounds. That is, positions for exterior rings must be ordered counterclockwise, and positions for holes must be ordered clockwise. Then, the coordinates field of a polygon must be an array of linear ring coordinate arrays, where the first must be the exterior ring, and any others must be interior rings. The exterior ring bounds the surface, and the interior rings (if present) bound holes within the surface. The example below shows a polygon with no holes. { "type" : "polygon", "coordinates" : [[[23.48, 35.16],[24.30, 35.16],[24.30, 35.50],[24.16, 35.61],[23.74, 35.70],[23.56, 35.60],[23.48, 35.16]]]} MultiPoint For type MultiPoint, the coordinates field is an array of two or more positions:{ "type" : "MultiPoint","coordinates" : [[-121.9447, 37.2975],[-121.9500, 37.3171],[-122.3975, 37.6144]]} MultiLineString For type MultiLineString, the coordinates member is an array of LineString coordinate arrays. { "type": "MultiLineString","coordinates": [[[100.0, 0.0], [01.0, 1.0]],[[102.0, 2.0], [103.0, 3.0]]] } MultiPolygon For type MultiPolygon, the coordinates member is an array of Polygon coordinate arrays. { "type": "MultiPolygon","coordinates": [[[[102.0, 2.0],[103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]], [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]]]] } GeometryCollection Instead of a coordinates field, a GeometryCollection has a geometries field. The value of geometries is an array. Each element of this array is a GeoJSON object whose kind is one of the six kinds defined above. In general, a GeometryCollection is a heterogeneous composition of smaller geometries. { "type": "GeometryCollection", "geometries": [{"type": "Point","coordinates": [100.0, 0.0]}, {"type": "LineString", "coordinates": [[101.0, 0.0], [102.0, 1.0]]}] } Page 23 The Oracle NoSQL Database has the following functions to use for searching GeoJSON data that has some relationship with a search geometry. boolean geo_intersect(any*, any*) boolean geo_inside(any*, any*) boolean geo_within_distance(any*, any*, double) boolean geo_near(any*, any*, double) In addition to the search functions, two other functions are available, and listed as the last two rows of the table: Function Type Details geo_intersect(any*, any*) boolean Raises an error at compile time if the function can detect that any operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: Returns false if any operand returns 0 or more than 1 items. Returns NULL if any operand returns NULL. Returns false if any operand returns an item that is not a valid GeoJson object. Finally, if both operands return a single GeoJson object, returns true if the two geometries have any points in common. Otherwise, returns false. geo_inside(any*, any*) boolean Raises an error at compile time if the function can detect that any operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: Returns false if any operand returns 0 or more than 1 item. Returns NULL if any operand returns NULL. Returns false if any operand returns an item that is not a valid GeoJson object. Finally, if both operands return a single GeoJson object and the second GeoJson is a polygon, the function returns true if the first geometry is completely contained inside the second polygon, with all of its points belonging to the interior of the polygon. The interior of a polygon is all the points in the polygon, except the points of the linear rings that define the polygons boundary. Otherwise, returns false. geo_within_distance(any*, any*, double) boolean Raises an error at compile time if the function detects that the first two operands will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: Returns false if any of the first two operands returns 0 or more than 1 item. Returns NULL if any of the first two operands returns NULL. Returns false if any of the first two operands returns an item that is not a valid GeoJson object. Finally, if both of the first two operands return a single GeoJson object, the function returns true if the first geometry is within a distance of N meters from the second geometry, where N is the number returned by the third operand. The distance between 2 geometries is defined as the minimum among the distances of any pair of points where the first point belongs to the first geometry, and the second point to the second geometry. Otherwise, returns false. geo_near(any*, any*, double) boolean The geo_near funcion is converted internally to a geo_within_distance function, with an an (implicit) order by the distance between the two geometries. However, if the query has an (explicit) order-by already, the function performs no ordering by distance. The geo_near function can appear only in the WHERE clause, and must be a top-level predicate. The geo_near function cannot be nested under an OR or NOT operator. geo_distance(any*, any*) double Raises an error at compile time if the function detects that an operand will not return a single valid GeoJson object. Otherwise, the runtime behavior is as follows: Returns -1 if any of the operands returns zero or more than 1 item. Returns -1 if any of the operands is not a geometry. Returns NULL if any operand returns NULL. Otherwise the function returns the geodetic distance between the 2 input geometries. The returned distance is the minimum among the distances of any pair of points, where the first point belongs to the first geometry and the second point to the second geometry. Between two such points, their distance is the length of the geodetic line that connects the points. geo_is_geometry(any*) boolean Returns false if an operand returns zero or more than 1 item. Returns NULL if an operand returns NULL. Returns true if the input is a single valid GeoJson object. Otherwise, false. Page 24 The SQL for Oracle NoSQL Database query processor can detect which of the existing indexes on a table can be used to optimize the execution of a query. This chapter provides a brief examples-based introduction to index creation, and queries using indexes. For a more detailed description of index creation and usage, see SQL Reference Guide. To make it possible to fit the example output on the page, the examples in this chapter use mode LINE. Page 25 This section builds on the examples that you began in Working with complex data. sql-> mode LINEQuery output mode is LINEsql-> create index idx_income on Persons (income);Statement completed successfullysql-> create index idx_age on Persons (age);Statement completed successfullysql-> SELECT * from PersonsWHERE income > 10000000 and age < 40; > Row 0 +-------------+--+ | id | 3 | +-------------+--+ | firstname | John | +-------------+--+ | lastname | Morgan | +-------------+--+ | age | 38 | +-------------+--+ | income | 100000000 | +-------------+--+ | lastLogin | 2016-11-29T08:21:35.4971 | +-------------+--+ | address | street | 187 Aspen Drive | | | city | Middleburg | | | state | FL | | | zipcode | NULL | | | phones | | | type | work | | | areacode | 305 | | | number | 1234079 | | | | | type | home | | | areacode | 305 | | | number | 2066401 | +-------------+--+ | connections | 1 | | | 4 | | | 2 | +-------------+--+ | expenses | food | 2000 | | | gas | 10 | | | travel | 700 | +-------------+--+1 row returned Page 26 In the previous section, both indexes are applicable. For index idx_income, the query condition income > 10000000 can be used as the starting point for an index scan that will retrieve only the index entries and associated table rows that satisfy this condition. Similarly, for index idx_age, the condition age < 40 can be used as the stopping point for the index scan. SQL for Oracle NoSQL Database has no way of knowing which of the 2 predicates is more selective, and it assigns the same "value" to each index, eventually picking the one whose name is first alphabetically. In the previous example, idx_age was used. To choose the idx_income index instead, the query should be written with an index hint: sql-> SELECT /*+ FORCE_INDEX(Persons idx_income) */ * from PersonsWHERE income > 10000000 and age < 40; > Row 0 +-------------+--+ | id | 3 | +-------------+--+ | firstname | John | +-------------+--+ | lastname | Morgan | +-------------+--+ | age | 38 | +-------------+--+ | income | 100000000 | +-------------+--+ | lastLogin | 2016-11-29T08:21:35.4971 | +-------------+--+ | address | street | 187 Aspen Drive | | | city | Middleburg | | | state | FL | | | zipcode | NULL | | | phones | | | type | work | | | areacode | 305 | | | number | 1234079 | | | | | | type | home | | | areacode | 305 | | | number | 2066401 | +-------------+--+ | connections | 1 | | | 4 | | | 2 | +-------------+--+ | expenses | food | 2000 | | | gas | 10 | | | travel | 700 | +-------------+--+1 row returned As shown above, hints are written as a special kind of comment that must be placed immediately after the SELECT keyword. What distinguishes a hint from a regular comment is the "+" character immediately after (without any space) the opening "/*". Summary: in this tutorial, you will learn how to use the Oracle FETCH clause to limit the rows returned by a query.Some RDBMS, such as MySQL and PostgreSQL have the LIMIT clause that allows you to retrieve a subset of rows generated by a query.See the following products and inventories tables in the sample database.The following query uses the LIMIT clause to get the top 5 products with the highest inventory level:SELECT product_name, quantityFROM inventories INNER JOIN products USING (product_id)ORDER BY quantity DESCLIMIT 5;Code language: SQL (Structured Query Language) (sql)In this example:First the ORDER BY clause sorts the products by stock quantity in descending order.Then, the LIMIT clause returns the first 5 products with the highest stock quantity.Oracle Database does not have the LIMIT clause. However, since the 12c release, it provided a similar but more flexible clause called the row limiting clause.By using the row-limiting clause, you can rewrite the query that uses the LIMIT clause above as follows:SELECT product_name, quantityFROM inventoriesINNER JOIN products USING(product_id)ORDER BY quantity DESC FETCH NEXT 5 ROWS ONLY;Code language: SQL (Structured Query Language) (sql)Try itIn this statement, the row limiting clause is:FETCH NEXT 5 ROWS ONLYCode language: SQL (Structured Query Language) (sql)Similar to the statement that uses LIMIT clause above, the row limiting clause returns the top 5 products with the highest inventory level.The following illustrates the syntax of the row limiting clause:[OFFSET offset ROWS]FETCH NEXT [row_count | percent PERCENT] ROWS [ONLY | WITH TIES]Code language: SQL (Structured Query Language) (sql)The OFFSET clause specifies the number of rows to skip before fetching rows. The OFFSET clause is optional. If you skip it, then the offset is 0, and FETCH clause will start fetching from the first row.The offset must be a number or an expression that evaluates to a number. It follows these rules:If the offset is negative, Oracle treats it as 0.If the offset is NULL or greater than the number of rows returned by the query, then the statement returns no rows.If the offset includes a fraction like 5.5, then the Oracle truncates the fractional portion.The FETCH clause specifies the number of rows or percentage of rows to return.For semantic clarity purposes, you can use the keyword ROWinstead ofROWS, FIRSTinstead ofNEXT. For example, the following clauses behave the same:FETCH NEXT 1 ROWSFETCH FIRST 1 ROWCode language: SQL (Structured Query Language) (sql)TheONLY returns exactly the number of rows or percentage of rows after FETCH NEXT (or FIRST).TheWITH TIESreturns additional rows with the same sort key as the last row fetched.Note that if you use WITH TIES, you must specify an ORDER BY clause in the query. If you dont, the query will not return the additional rows.Since Oracle stores rows in a table in an unspecified order, you should always use the FETCH clause with the ORDER BY clause to get a predictable result.Lets take an example of using the FETCH clause.The following statement uses the FETCH clause to retrieve the top 10 products with the highest quantity:SELECT product_name, quantityFROM inventoriesINNER JOIN products USING(product_id)ORDER BY quantity DESC FETCH NEXT 10 ROWS ONLY;Code language: SQL (Structured Query Language) (sql)Try itHow the query works:First, the ORDER BY clause sorts rows by quantity from high to low.Second, the FECTCH clause retrieves the top 10 rows.The following statement uses the FETCH clause to retrieve the top 3 products with the highest standard cost in category 1:SELECT product_name, quantityFROM inventories INNER JOIN products USING (product_id)WHERE category_id = 1ORDER BY quantity DESCFETCH NEXT 3 ROWS ONLY;Try itOutput:The following query uses the FETCH clause to with the WITH TIES option to retrieve the top 10 rows with ties:SELECT product_name, quantityFROM inventories INNER JOIN products USING (product_id)ORDER BY quantity DESCFETCH NEXT 10 ROWSWITH TIES;Code language: SQL (Structured Query Language) (sql)Try itAlthough the statement tells Oracle to return 10 rows, it returns 12 rows.The reason is the WITH TIES option, and there are two additional rows with the same quantity (273).The following query uses the FETCH clause to retrieve the top 5% products with the highest quantity:SELECT product_name, quantityFROM inventories INNER JOIN products USING (product_id)ORDER BY quantity DESCFETCH FIRST 5 PERCENT ROWS ONLY;Code language: SQL (Structured Query Language) (sql)Try itThe inventories table has 1,112 rows. Therefore, 5% of 1,112is 55.6 which is rounded up to 56 rows.The following query uses the OFFSET clause to skip the first 10 rows with the highest quantity and return the next 10 ones:SELECT product_name, quantityFROM inventories INNER JOIN products USING (product_id)ORDER BY quantity DESCOFFSET 10 ROWSFETCH NEXT 10 ROWS ONLY;Code language: SQL (Structured Query Language) (sql)Try itHow the query works:First, the ORDER BY clause sorts rows by quantity from high to low.Second, the OFFSET clause skips the first 10 rows.Third, the FECTCH clause retrieves the next 10 rows.Use the Oracle FETCH clause to limit the number of rows returned by a query.Was this tutorial helpful? View All Tutorials Login to Run Tutorial Tutorial Sorting and Limiting Rows: Databases for Developers Description An introduction to sorting data with order by and restricting rows to the top N. Tags order by, fetch first Area SQL General Contributor Chris Saxon (Oracle) Created Thursday May 03, 2018 Modules 13 create table toys (toy_name varchar2(30), weight integer, price number(5,2), purchased_date date, last_lost_date date);insert into toys values ('Miss Snuggles', 4, 9.99, date'2018-02-01', date'2018-06-01');insert into toys values ('Baby Turtle', 1, 5.00, date'2016-09-01', date'2017-03-03');insert into toys values ('Kangaroo', 10, 29.99, date'2017-03-01', date'2018-06-01');insert into toys values ('Blue Dinosaur', 8, 9.99, date'2013-07-01', date'2016-11-01');insert into toys values ('Purple Ninja', 8, 29.99, date'2018-02-01', null);commit; Relational database programming languageThis article is about the database language. For other uses, see SQL (disambiguation)."SEQUEL" redirects here. For the topic of the word, see sequel. For other uses, see Sequel (disambiguation).SQL (Structured Query Language)ParadigmDeclarativeFamilyQuery languageDesignedbyDonald D. ChamberlinRaymond F. BoyceDeveloperISO/IEC JTC1 (Joint Technical Committee1) / SC32 (Subcommittee32) / WG3 (Working Group3)Firstappeared1973; 52years ago(1973)Stable releaseSQL:2023 / June2023; 2years ago(2023-06)Typing disciplineStatic, strongOSCross-platformWebsitewww.iso.org/standard/76583.htmlMajor implementationsManyDialectsSQL-86SQL-89SQL-92SQL:1999SQL:2003SQL:2006SQL:2008SQL:2011SQL:2016SQL:2023Influenced byDatalogInfluencedCQL, LINQ, SPARQL, SOQL, PowerShell,[1] JPQL, jOOQ, N1QL, GQL Structured Query Language at WikibooksSQL (file format)Filename extension.sqlInternet mediatypeapplication/sql[2][3]DevelopedbyISO/IECInitial release1986; 39years ago(1986)Type of formatDatabaseStandardISO/IEC 9075Open format?YesWebsitewww.iso.org/standard/76583.htmlStructured Query Language (SQL) (pronounced /skjul/ S-Q-L; or alternatively as /sikwl/ "sequel")[4][5] is a domain-specific language used to manage data, especially in a relational database management system (RDBMS). It is particularly useful in handling structured data, i.e., data incorporating relations among entities and variables.Introduced in the 1970s, SQL offered two main advantages over older readwrite APIs such as ISAM or VSAM. Firstly, it introduced the concept of accessing many records with one single command. Secondly, it eliminates the need to specify how to reach a record, i.e., with or without an index.Originally based upon relational algebra and tuple relational calculus, SQL consists of many types of statements,[6] which may be informally classed as sublanguages, commonly: data query language (DQL), data definition language (DDL), data control language (DCL), and data manipulation language (DML).[7]The scope of SQL includes data query, data manipulation (insert, update, and delete), data definition (schema creation and modification), and data access control. Although SQL is essentially a declarative language (4GL), it also includes procedural elements.SQL was one of the first commercial languages to use Edgar F. Codd's relational model. The model was described in his influential 1970 paper, "A Relational Model of Data for Large Shared Data Banks".[8] Despite not entirely adhering to the relational model as described by Codd, SQL became the most widely used database language.[9][10]SQL became a standard of the American National Standards Institute (ANSI) in 1986 and of the International Organization for Standardization (ISO) in 1987.[11] Since then, the standard has been revised multiple times to include a larger set of features and incorporate common extensions. Despite the existence of standards, virtually no implementations in existence adhere to it fully, and most SQL code requires at least some changes before being ported to different database systems.SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce after learning about the relational model from Edgar F. Codd[12] in the early 1970s.[13] This version, initially called SEQUEL (Structured English Query Language), was designed to manipulate and retrieve data stored in IBM's original quasirelational database management system, System R, which a group at IBM San Jose Research Laboratory had developed during the 1970s.[13]Chamberlin and Boyce's first attempt at a relational database language was SQUARE (Specifying Queries in A Relational Environment), but it was difficult to use due to subscript/superscript notation. After moving to the San Jose Research Laboratory in 1973, they began work on a sequel to SQUARE.[12] The original name SEQUEL, which is widely regarded as a pun on QUEL, the query language of Ingres,[14] was later changed to SQL (dropping the vowels) because "SEQUEL" was a trademark of the UK-based Hawker Siddeley Dynamics Engineering Limited company.[15] The label SQL later became the acronym for Structured Query Language.[16]After testing SQL at customer test sites to determine the usefulness and practicality of the system, IBM began developing commercial products based on their System R prototype, including System/38, SQL/DS, and IBM Db2, which were commercially available in 1979, 1981, and 1983, respectively.[17] IBM's endorsement caused the industry to move to SQL from alternatives like QUEL.[18]In the late 1970s, Relational Software, Inc. (now Oracle Corporation) saw the potential of the concepts described by Codd, Chamberlin, and Boyce, and developed their own SQL-based RDBMS with aspirations of selling it to the U.S. Navy, Central Intelligence Agency, and other U.S. government agencies. In June 1979, Relational Software introduced one of the first commercially available implementations of SQL, Oracle V2 (Version2) for VAX computers.By 1986, ANSI and ISO standard groups officially adopted the standard "Database Language SQL" language definition. New versions of the standard were published in 1989, 1992, 1996, 1999, 2003, 2006, 2008, 2011,[12] 2016 and most recently, 2023.[19]SQL language revisionsSQL-86SQL-89SQL-92SQL:1999SQL:2003SQL:2006SQL:2008SQL:2011SQL:2016SQL:2019SQL:2023vteSQL implementations are

incompatible between vendors and do not necessarily completely follow standards. In particular, date and time syntax, string concatenation, NULLs, and comparison case sensitivity vary from vendor to vendor. PostgreSQL[20] and Mimer SQL[21] strive for standards compliance, though PostgreSQL does not adhere to the standard in all cases. For example, the folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard,[22] which says that unquoted names should be folded to upper case.[3] Thus, according to the standard, Foo should be equivalent to FOO, not foo.Popular implementations of SQL commonly omit support for basic features of Standard SQL, such as the DATE or TIME data types. The most obvious such examples, and incidentally the most popular commercial and proprietary SQL DBMSs, are Oracle (whose DATE behaves as DATETIME,[24][25] and lacks a TIME type)[26] and MS SQL Server (before the 2008 version). As a result, SQL code can rarely be ported between database systems without modifications.Several reasons for the lack of portability between database systems include:The complexity and size of the SQL standard means that most implementers do not support the entire standard.The SQL standard does not specify the database behavior in some important areas (e.g., indices, file storage), leaving implementations to decide how to behave.The SQL standard defers some decisions to individual implementations, such as how to name a results column that was not named explicitly.[27]:207The SQL standard precisely specifies the syntax that a conforming database system must implement. However, the standard's specification of the semantics of language constructs is less well-defined, leading to ambiguity.Many database vendors have large existing customer bases; where the newer version of the SQL standard conflicts with the prior behavior of the vendor's database, the vendor may be unwilling to break backward compatibility.Little commercial incentive exists for vendors to make changing database suppliers easier (see vendor lock-in).Users evaluating database software tend to place other factors such as performance higher in their priorities than standards conformance.SQL was adopted as a standard by the ANSI in 1986 as SQL-86[28] and the ISO in 1987.[11] It is maintained by ISO/IEC JTC 1, Information technology, Subcommittee SC 32, Data management and interchange.Until 1996, the National Institute of Standards and Technology (NIST) data-management standards program certified SQL DBMS compliance with the SQL standard. Vendors now self-certify the compliance of their products.[29]The original standard declared that the official pronunciation for "SQL" was an initialism: /skjul/ ("ess cue el").[9] Regardless, many English-speaking database professionals (including Donald Chamberlin himself[30]) use the acronym-like pronunciation of /sikwl/ ("sequel"),[31] mirroring the language's prerelease development name, "SEQUEL".[13][15][30] The SQL standard has gone through a number of revisions:Timeline of SQL languageYearOfficial standardInformalnameComments19861987ANSI X3.135:1986ISO/IEC 9075:1987FIPS PUB 127SQL-86SQL-87First formalized by ANSI, adopted as FIPS PUB 1271989ANSI X3.135-1989ISO/IEC 9075:1989FIPS PUB 127-1SQL-89Minor revision that added integrity constraints, adopted as FIPS PUB 127-11992ANSI X3.135-1992ISO/IEC 9075:1992FIPS PUB 127-2SQL-92SQL2Major revision (ISO 9075), Entry Level SQL-92, adopted as FIPS PUB 127-21999ISO/IEC 9075:1999SQL:1999SQL3Added regular expression matching, recursive queries (e.g., transitive closure), triggers, support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g., structured types), support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT)2003ISO/IEC 9075:2003SQL:2003Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with autogenerated values (including identity columns)2006ISO/IEC 9075-14:2006SQL:2006Adds Part 14, defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[32]2008ISO/IEC 9075:2008SQL:2008Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[33] FETCH clause2011ISO/IEC 9075:2011SQL:2011Adds temporal data (PERIOD FOR)[34] (more information at Temporal database#History). Enhancements for window functions and FETCH clause.[35]2016ISO/IEC 9075:2016SQL:2016Adds row pattern matching, polymorphic table functions, operations on JSON data stored in character string fields2019ISO/IEC 9075-15:2019SQL:2019Adds Part 15, multidimensional arrays (MDarray type and operators)2023ISO/IEC 9075:2023SQL:2023Adds data type JSON (SQL/Foundation); Adds Part 16, Property Graph Queries (SQL/PGQ)The standard is commonly denoted by the pattern: ISO/IEC 9075-n:yyyy Part n: title, or, as a shortcut, ISO/IEC 9075. Interested parties may purchase the standards documents from ISO,[36] IEC, or ANSI. Some old drafts are freely available.[37][38]ISO/IEC 9075 is complemented by ISO/IEC 13249: SQL Multimedia and Application Packages and some Technical reports.Main article: SQL syntax U P D A T E c l a u s e : { U P D A T E c o u n t r i e s t a b l e } S E T c l a u s e : { S E T p o p u l a t i o n c o l u m n = p o p u l a t i o n + 1 l i t e r a l e x p r e s s i o n } W H E R E c l a u s e : { W H E R E n a m e c o l u m n = U S A l i t e r a l e x p r e s s i o n p r e d i c a t e } ; } statement S Q L q u e r y {\displaystyle \underbrace {\left.{\begin{array}{rl}\textstyle {\mathtt {UPDATE~clause{\mathtt {:}}}}&\{{\mathtt {UPDATE\ \overbrace {\mathtt {countries}} ^{\mathtt {table}}}}\}\\\textstyle {\mathtt {SET~clause:}}&\{{{\mathtt {SET\ \overbrace {\mathtt {population}} ^{\mathtt {column}}=~}}\overbrace {\mathtt {{population}+\underbrace {\mathtt {1}} _{\mathtt {literal}}}} ^{\mathtt {expression}}}\}\\\textstyle {\mathtt {WHERE~clause:}}&\{{{\mathtt {WHERE\ \underbrace {\overbrace {\mathtt {name}} ^{\mathtt {column}}{\mathtt {=}}\overbrace {\overbrace {\mathtt {'USA'}} ^{\mathtt {literal}}} ^{\mathtt {expression}}} _{\mathtt {predicate}}}}\}{\textstyle {\texttt {;}}}\end{array}}\right\}} {\textstyle {\texttt {statement}}}} _{\textstyle {\mathtt {SQL~query}}}} A chart showing several of the SQL language elements comprising a single statementThe SQL language is subdivided into several language elements, including:Clauses, which are constituent components of statements and queries. (In some cases, these are optional.)[39]Expressions, which can produce either scalar values, or tables consisting of columns and rows of dataPredicates, which specify conditions that can be evaluated to SQL three-valued logic (3VL) (true/false/unknown) or Boolean truth values and are used to limit the effects of statements and queries, or to change program flow.Queries, which retrieve the data based on specific criteria. This is an important element of SQL.Statements, which may have a persistent effect on schemata and data, or may control transactions, program flow, connections, sessions, or diagnostics.SQL statements also include the semicolon (";") statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.Insignificant whitespace is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.SQL is designed for a specific purpose: to query data contained in a relational database. SQL is a set-based, declarative programming language, not an imperative programming language like C or BASIC. However, extensions to Standard SQL add procedural programming language functionality, such as control-of-flow constructs.In addition to the standard SQL/PSM extensions and proprietary SQL extensions, procedural and object-oriented programmability is available on many SQL platforms via DBMS integration with other languages. The SQL standard defines SQL/JRT extensions (SQL Routines and Types for the Java Programming Language) to support Java code in SQL databases. Microsoft SQL Server 2005 uses the SQLCLR (SQL Server Common Language Runtime) to host managed .NET assemblies in the database, while prior versions of SQL Server were restricted to unmanaged extended stored procedures primarily written in C. PostgreSQL lets users write functions in a wide variety of languagesincluding Perl, Python, Tcl, JavaScript (PL/V8) and C.[40]A distinction should be made between alternatives to SQL as a language, and alternatives to the relational model itself. Below are proposed relational alternatives to the SQL language. See navigational database and NoSQL for alternatives to the relational model..QL: object-oriented Datalog4D Query Language (4D QL)Datalog: critics suggest that Datalog has two advantages over SQL: it has cleaner semantics, which facilitates program understanding and maintenance, and it is more expressive, in particular for recursive queries.[41]HTSQL: URL based query methodIBM Business System 12 (IBM BS12): one of the first fully relational database management systems, introduced in 1982ISBLjOOQ: SQL implemented in Java as an internal domain-specific languageJava Persistence Query Language (JPQL): The query language used by the Java Persistence API and Hibernate persistence libraryJavaScript: MongoDB implements its query language in a JavaScript API.LINQ: Runs SQL statements written like language constructs to query collections directly from inside .Net codeObject Query LanguageQBE (Query By Example) created by Mosh Zloof, IBM 1977QUEL introduced in 1974 by the U.C. Berkeley Ingres project, closer to tuple relational calculus than SQLXQueryDistributed Relational Database Architecture (DRDA) was designed by a workgroup within IBM from 1988 to 1994. DRDA enables network-connected relational databases to cooperate to fulfill SQ requests.[42][43]An interactive user or program can issue SQL statements to a local RDB and receive tables of data and status indicators in reply from remote RDBs. SQL statements can also be compiled and stored in remote RDBs as packages and then invoked by package name. This is important for the efficient operation of application programs that issue complex, high-frequency queries. It is especially important when the tables to be accessed are located in remote systems.The messages, protocols, and structural components of DRDA are defined by the Distributed Data Management Architecture. Distributed SQL processing ala DRDA is distinctive from contemporary distributed SQL databases.SQL deviates in several ways from its theoretical foundation, the relational model and its tuple calculus. In that model, a table is a set of tuples, while in SQL, tables and query results are lists of rows; the same row may occur multiple times, and the order of rows can be employed in queries (e.g., in the LIMIT clause).Critics argue that SQL should be replaced with a language that returns strictly to the original foundation: for example, see The Third Manifesto by Hugh Darwen and C.J. Date (2006, ISBN0-321-39942-0).Early specifications did not support major features, such as primary keys. Result sets could not be named, and subqueries had not been defined. These were added in 1992.[12]The lack of sum types has been described as a roadblock to full use of SQL's user-defined types. JSON support, for example, needed to be added by a new standard in 2016.[44]The concept of Null is the subject of some debate. The Null marker indicates the absence of a value, and is distinct from a value of 0 for an integer column or an empty string for a text column. The concept of Nulls enforces the 3-valued-logic in SQL, which is a concrete implementation of the general 3-valued logic.[12]Another popular criticism is that it allows duplicate rows, making integration with languages such as Python, whose data types might make accurately representing the data difficult,[12] in terms of parsing and by the absence of modularity. This is usually avoided by declaring a primary key, or a unique constraint, with one or more columns that uniquely identify a row in the table.In a sense similar to objectrelational impedance mismatch, a mismatch occurs between the declarative SQL language and the procedural languages in which SQL is typically embedded.[citation needed]The SQL standard defines three kinds of data types (chapter 4.1.1 of SQL/Foundation):predefined data typesconstructed typesuser-defined types.Constructed types are one of ARRAY, MULTISET, REF(erence), or ROW. User-defined types are comparable to classes in object-oriented language with their own constructors, observers, mutators, methods, inheritance, overloading, overwriting, interfaces, and so on. Predefined data types are intrinsically supported by the implementation.Character typesCharacter (CHAR)Character varying (VARCHAR)Character large object (CLOB)National character typesNational character (NCHAR)National character varying (NCHAR VARYING)National character large object (NCLOB)Binary typesBinary (BINARY)Binary varying (VARBINARY)Binary large object (BLOB)Numeric typesExact numeric types (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT)Approximate numeric types (FLOAT, REAL, DOUBLE PRECISION)Decimal floating-point type (DECFLOAT)Datetime types (DATE, TIME, TIMESTAMP)Interval type (INTERVAL)BooleanXML (see SQL/XML)[45]JSONObject databaseList of relational database management systemsComparison of relational database management systemsComparison of objectrelational database management systemsQuery by ExampleSQL reserved wordsSQL syntaxOracle PL/SQLMicrosoft Transact-SQL (T-SQL)Online transaction processing (OLTP)Online analytical processing (OLAP)Data warehouseRelational data stream management systemNoSQLMUMPSHierarchical database modelStar schemaSnowflake schema^ Paul, Ryan (24 October 2005). "A guided tour of the Microsoft Command Shell". Ars Technica. Retrieved 10 April 2011.^ "Media Type registration for application/sql". Internet Assigned Numbers Authority. 10 April 2013. Archived from the original on 7 May 2013. Retrieved 10 April 2013.^ Shafranovich, Y. (April 2013). "The application/sql Media Type, RFC 6922". Internet Engineering Task Force. p.3. doi:10.17487/RFC6922. Archived from the original on 13 October 2014. Retrieved 10 April 2013.^ Beaulieu, Alan (April 2009). Mary E Treseler (ed.). Learning SQL (2nded.). Sebastopol, CA, USA: O'Reilly. ISBN978-0-596-52083-0.^ Chamberlin, Donald D.; Frana, Philip L. (3 October 2001). "Oral history interview with Donald D. Chamberlin". University Digital Conservancy. hdl:11299/107215. Archived from the original on 24 February 2020. Retrieved 14 January 2020. We changed the original name "SEQUEL" to SQL because we got a letter from somebody's lawyer that said the name "SEQUEL" belonged to them. We shortened it to SQL, for Structured Query Language, and the product was known as SQL/DS.^ SQL-92, 4.22 SQL-statements, 4.22.1 Classes of SQL-statements "There are at least five ways of classifying SQL-statements:", 4.22.2, SQL statements classified by function "The following are the main classes of SQL-statements:"; SQL:2003 4.11 SQL-statements, and later revisions.^ Chatham, Mark (2012). Structured Query Language By Example - Volume I: Data Query Language. Lulu.com. p.8. ISBN9781291199512.^ Codd, Edgar F. (June 1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM. 13 (6): 37787. CiteSeerX10.1.1.88.646. doi:10.1145/362384.362685. S2CID207549016.^ a b Chapple, Mike. "SQL Fundamentals". Databases. About.com. Archived from the original on 22 February 2009. Retrieved 28 January 2009.^ "Structured Query Language (SQL)". International Business Machines. 27 October 2006. Retrieved 10 June 2007.^ a b "ISO 9075:1987: Information technology Database languages SQL Part 1: Framework (SQL/Framework)". 1 June 2023. Retrieved 24 June 2025.{{cite web}}: CS1 maint: url-status (link)^ a b c d e f Chamberlin, Donald (2012). "Early History of SQL". IEEE Annals of the History of Computing. 34 (4): 7882. doi:10.1109/MAHC.2012.61. S2CID1322572.^ a b c Chamberlin, Donald D; Boyce, Raymond F (1974). "SEQUEL: A Structured English Query Language" (PDF). Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control. Association for Computing Machinery: 24964. Archived from the original (PDF) on 26 September 2007. Retrieved 9 June 2007.^ Starkey, Jim. "Dynamic SQL, Plumbing, and the Internal API". www.ibphoenix.com. Archived from the original on 19 January 2023. Retrieved 19 January 2023.^ a b Oppel, Andy (27 February 2004). Databases Demystified. San Francisco, CA: McGraw-Hill Osborne Media. pp.901. ISBN978-0-07-146960-9. Archived from the original on 14 January 2012. Retrieved 12 May 2011.^ Yasar, Kinza; Loshin, Peter; Sirkin, Jessica. "What is Structured Query Language (SQL)? | Definition from TechTarget". Search Data Management. Informa TechTarget. Archived from the original on 10 March 2025. Retrieved 25 March 2025.^ "History of IBM, 1978". IBM Archives. IBM. 23 January 2003. Archived from the original on 17 January 2005. Retrieved 9 June 2007.^ Morgenthaler, Gary (8 December 2005). "Oral History of Gary Morgenthaler" (PDF) (Interview). Interviewed by Luann Johnson. Computer History Museum. p.18. Retrieved 30 May 2025.^ "ISO - ISO/IEC JTC 1/SC 32 - Data management and interchange". www.iso.org. Archived from the original on 16 March 2017. Retrieved 2 January 2021.^ "About PostgreSQL". PostgreSQL 9.1 official website. PostgreSQL Global Development Group. 2012. Archived from the original on 9 March 2012. Retrieved 9 March 2012. PostgreSQL prides itself in standards compliance. Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard^ "Mimer SQL, Built on Standards". Mimer SQL official website. Mimer Information Technology. 2009. Archived from the original on 3 May 2016. Retrieved 11 December 2018.^ "4.1. Lexical Structure". PostgreSQL documentation. 2018. Archived from the original on 5 October 2018. Retrieved 11 December 2018.^ "(Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL, Section 5.2, syntax rule 11". 30 July 1992. Archived from on 21 June 2006. Retrieved 8 October 2009.^ Lorentz, Diana; Roeser, Mary Beth; Abraham, Sundeep; Amor, Angela; Arora, Geeta; Arora, Vikas; Ashdown, Lance; Baer, Hermann; Bellamkonda, Shrikanth (October 2010) [1996]. "Basic Elements of Oracle SQL: Data Types". Oracle Database SQL Language Reference 11g Release 2 (11.2). Oracle Database Documentation Library. Redwood City, CA: Oracle USA, Inc. Archived from the original on 20 November 2010. Retrieved 29 December 2010. For each DATE value, Oracle stores the following information: century, year, month, date, hour, minute, and second^ Lorentz, Diana; Roeser, Mary Beth; Abraham, Sundeep; Amor, Angela; Arora, Geeta; Arora, Vikas; Ashdown, Lance; Baer, Hermann; Bellamkonda, Shrikanth (October 2010) [1996]. "Basic Elements of Oracle SQL: Data Types". Oracle Database SQL Language Reference 11g Release 2 (11.2). Oracle Database Documentation Library. Redwood City, CA: Oracle USA, Inc. Archived from the original on 20 November 2010. Retrieved 29 December 2010. The datetime data types are DATE...^ Lorentz, Diana; Roeser, Mary Beth; Abraham, Sundeep; Amor, Angela; Arora, Geeta; Arora, Vikas; Ashdown, Lance; Baer, Hermann; Bellamkonda, Shrikanth (October 2010) [1996]. "Basic Elements of Oracle SQL: Data Types". Oracle Database SQL Language Reference 11g Release 2 (11.2). Oracle Database Documentation Library. Redwood City, CA: Oracle USA, Inc. Archived from the original on 20 November 2010. Retrieved 29 December 2010. Do not define columns with the following SQL/DS and DB2 data types, because they have no corresponding Oracle data type:... TIME^ Date, Chris J. (2013). Relational Theory for Computer Professionals: What Relational Databases are Really All About (1.ed.). Sebastopol, Calif: O'Reilly Media. ISBN978-1-449-36943-9.^ "Finding Aid". X3H2 Records, 197895. American National Standards Institute. Archived from the original on 19 June 2010. Retrieved 26 April 2008.^ Doll, Shelley (19 June 2002). "Is SQL a Standard Anymore?". TechRepublic's Builder.com. TechRepublic. Archived from the original on 5 July 2012. Retrieved 12 April 2016.^ a b Gillespie, Patrick. "Pronouncing SQL: S-Q-L or Sequel?". Archived from the original on 27 May 2012. Retrieved 12 February 2012.^ Melton, Jim; Alan R Simon (1993). "1.2. What is SQL?". Understanding the New SQL: A Complete Guide. Morgan Kaufmann. p.536. ISBN978-1-55860-245-8. SQL (correctly pronounced "ess cue ell," instead of the somewhat common "sequel")...^ Wagner, Michael (2010). SQL/XML:2006 - Evaluierung der Standardkonformitt ausgewhlter Datenbanksysteme. Diplomica Verlag. p.100. ISBN978-3-8366-9609-8.^ "SQL:2008 now an approved ISO international standard". Sybase. July 2008. Archived from the original on 28 June 2011.^ Krishna Kulkarni, Jan-Eike Michels (September 2012). "Temporal features in SQL:2011" (PDF). SIGMOD Record. 41 (3).^ Fred Zemke (2012). "What's new in SQL:2011" (PDF). Oracle Corporation.^ "ISO/IEC 9075". Archived from the original on 12 August 2022. Retrieved 9 August 2022.^ SQL:1992 draft (text), archived from the original on 21 June 2006, retrieved 8 October 2009^ SQL:2011 draft, Whitemarsh Information Systems Corporation, archived from the original (Zip) on 25 May 2023^ ANSI/ISO/IEC International Standard (IS). Database Language SQLPart 2: Foundation (SQL/Foundation). 1999.^ "PostgreSQL server programming". PostgreSQL 9.1 official documentation. postgresql.org. 2011. Archived from the original on 2 March 2012. Retrieved 9 March 2012.^ Fernando Saenz-Perez. "Outer Joins in a Deductive Database System" (PDF). Lbd.udc.es. Archived (PDF) from the original on 9 August 2016. Retrieved 16 January 2017.^ Reinsch, R. (1988). "Distributed database for SAA". IBM Systems Journal. 27 (3): 362389. doi:10.1147/sj.273.0362.^ Distributed Relational Database Architecture Reference. IBM Corp. SC26-4651-0. 1990.^ Brandon, Jamie (July 2021). "Against SQL". Retrieved 2 August 2021.^ "SQL 2003 Standard Support in Oracle Database 10g" (PDF). Oracle. Oracle Corporation. November 2003. Archived (PDF) from the original on 27 March 2024. Retrieved 27 March 2024. XML supported was added in ANSI SQL 2003, part 14.Codd, Edgar F (June 1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM. 13 (6): 37787. doi:10.1145/362384.362685. S2CID207549016.Discussion on alleged SQL flaws (C2 wiki)C. J. Date with Hugh Darwen: A Guide to the SQL standard: a users guide to the standard database language SQL, 4th ed., Addison Wesley, USA 1997, ISBN978-0-201-96426-41995 SQ Reunion: People, Projects, and Politics, by Paul McJones (ed.): transcript of a reunion meeting devoted to the personal history of relational databases and SQL.American National Standards Institute. X3H2 Records, 19781995 Charles Babbage Institute Collection documents the H2 committee's development of the NDL and SQL standards.Oral history interview with Donald D. Chamberlin Charles Babbage Institute In this oral history Chamberlin recounts his early life, his education at Harvey Mudd College and Stanford University, and his work on relational database technology. Chamberlin was a member of the System R research team and, with Raymond F. Boyce, developed the SQL database language. Chamberlin also briefly discusses his more recent research on XML query languages.Portal: Computer programmingSQL at Wikipedia's sister projects:Media from CommonsTextbooks from WikibooksResources from WikiversityData from WikidataRetrieved from " 2Group of computer programming languagesA fourth-generation programming language (4GL) is a high-level computer programming language that belongs to a class of languages envisioned as an advancement upon third-generation programming languages (3GL). Each of the programming language generations aims to provide a higher level of abstraction of the internal computer hardware details, making the language more programmer-friendly, powerful, and versatile. While the definition of 4GL has changed over time, it can be typified by operating more with large collections of information at once rather than focusing on just bits and bytes. Languages claimed to be 4GL may include support for database management, report generation, mathematical optimization, graphical user interface (GUI) development, or web development. Some researchers state that 4GLs are a subset of domain-specific languages.[1][2]The concept of 4GL was developed from the 1970s through the 1990s, overlapping most of the development of 3GL, with 4GLs identified as "non-procedural" or "program-generating" languages, contrasted with 3GLs being algorithmic or procedural languages. While 3GLs like C, C++, C#, Java, and JavaScript remain popular for a wide variety of uses, 4GLs as originally defined found uses focused on databases, reports, and websites.[3] Some advanced 3GLs like Python, Ruby, and Perl combine some 4GL abilities within a general-purpose 3GL environment,[4] and libraries with 4GL-like features have been developed as add-ons for most popular 3GLs, producing languages that are a mix of 3GL and 4GL, blurring the distinction.[5]In the 1980s and 1990s, there were efforts to develop fifth-generation programming languages (5GL).Though used earlier in papers and discussions, the term 4GL was first used formally by James Martin in his 1981 book Application Development Without Programmers[6] to refer to non-procedural, high-level specification languages. In some primitive way, early 4GLs were included in the Informatics MARK-IV (1967) product and Sperry's MAPPER (1969 internal use, 1979 release).The motivations for the '4GL' inception and continued interest are several. The term can apply to a large set of software products. It can also apply to an approach that looks for greater semantic properties and implementation power. Just as the 3GL offered greater power to the programmer, so too did the 4GL open up the development environment to a wider population.The early input scheme for the 4GL supported entry of data within the 72-character limit of the punched card (8 bytes used for sequencing) where a card's tag would identify the type or function. With judicious use of a few cards, the 4GL deck could offer a wide variety of processing and reporting capability whereas the equivalent functionality coded in a 3GL could subsume, perhaps, a whole box or more of cards.[7]The 72-character format continued for a while as hardware progressed to larger memory and terminal interfaces. Even with its limitations, this approach supported highly sophisticated applications.As interfaces improved and allowed longer statement lengths and grammar-driven input handling, greater power ensued. An example of this is described on the Nomad page.Another example of Nomad's power is illustrated by Nicholas Rawlings in his comments for the Computer History Museum about NCSS (see citation below). He reports that James Martin asked Rawlings for a Nomad solution to a standard problem Martin called the Engineer's Problem: "give 6% raises to engineers whose job ratings had an average of 7 or better." Martin provided a "dozen pages of COBOL, and then just a page or two of Mark IV, from Informatics." Rawlings offered the following single statement, performing a set-at-a-time operation...The development of the 4GL was influenced by several factors, with the hardware and operating system constraints having a large weight. When the 4GL was first introduced, a disparate mix of hardware and operating systems mandated custom application development support that was specific to the system in order to ensure sales. One example is the MAPPER system developed by Sperry. Though it has roots back to the beginning, the system has proven successful in many applications and has been ported to modern platforms. The latest variant is embedded in the BIS[8] offering of Unisys. MARK-IV is now known as VISION:BUILDER and is offered by Computer Associates.The Santa Fe railroad used MAPPER to develop a system in a project that was an early example of 4GL, rapid prototyping, and programming by users.[9] The idea was that it was easier to teach railroad experts to use MAPPER than to teach programmers the "intricacies of railroad operations".[10]One of the early (and portable) languages that had 4GL properties was RAMIS developed by Gerald C. Cohen at Mathematica, a mathematical software company. Cohen left Mathematica and founded Information Builders to create a similar reporting-oriented 4GL, called FOCUS.Later 4GL types are tied to a database system and are far different from the earlier types in their use of techniques and resources that have resulted from the general improvement of computing with time.An interesting twist to the 4GL scene is realization that graphical interfaces and the related reasoning done by the user form a 'language' that is poorly understood.A number of different types of 4GLs exist:Table-driven (codeless) programming, usually running with a runtime framework and libraries. Instead of using code, the developer defines their logic by selecting an operation in a pre-defined list of memory or data table manipulation commands. In other words, instead of coding, the developer uses table-driven algorithm programming (see also control tables that can be used for this purpose). These types of tools can be used for business application development usually consisting in a package allowing for both business data manipulation and reporting, therefore they come with GUI screens and report editors. They usually offer integration with one or more lower level dynamic-link library (DLL), generated from a typical 3GL for when the need arises for more hardware/OS specific operations.Report generator programming languages take a description of the data format and the report to generate and from that they either generate the required report directly or they generate a program to generate the report (see also RPG).Similarly, screen generators (also called forms generators) manage online interactions with the application system users or generate programs to do so.More ambitious 4GLs (sometimes termed fourth generation environments) attempt to automatically generate whole systems from the outputs of computer-aided software engineering (CASE) tools, specifications of screens and reports, and possibly also the specification of some additional processing logic.Data management 4GLs such as SAS, SPSS, and Stata provide sophisticated coding commands for data manipulation, file reshaping, case selection, and data documentation in the preparation of data for statistical analysis and reporting.So-called "XTalk" languages, developed initially with Apple's Hypercard in 1987. Hypercard was the progenitor of more modern and powerful programs such as SuperCard, Toolbook and LiveCode.Some 4GLs have integrated tools that allow for the easy specification of all the required information:James Martin's version of data engineering systems development methodology was automated to allow the input of the results of system analysis and design in the form of data flow diagrams, entity relationship diagrams, entity life history diagrams etc., from which hundreds of thousands of lines of COBOL would be generated overnight.More recently Oracle Corporation's Oracle Designer and Oracle Developer Suite 4GL products could be integrated to produce database definitions and the forms and reports programs.In the twenty-first century, 4GL systems have emerged as "low code" environments or platforms for the problem of rapid application development in short periods of time. Vendors often provide sample systems such as CRM, contract management, bug tracking from which development can occur with little programming.[11]4th DimensionAccell/SQL (4GL) from Unify Corporation.Broadcom 2e Formerly Synon 4GL RPG/COBOL GeneratorCA-Telon 4GL Cobol/PLI generatorClarionClipperCognos PowerHouse 4GLDataFlex (Microsoft Windows, web, mobile)DATATRIEVEdBaseFileMakerFOCUSFort TOOL (transactional object-oriented language)GeneXusHarbourHyperCard (development and support were ended)IBM Rational EGL (Enterprise Generation Language)InforLabVIEWLANSALINCLiveCodeM-PowerNATURALOmnis Studio SDKOracle Application Development FrameworkOutSystems (Productivity/PaaS)Jmix/CUBA Platform (Productivity/Framework)PowerBuilderR:BaseSheerPower4GL (Microsoft Windows only)SQLWindows/Team DeveloperUnifaceUnix ShellVisual FoxPro (development and support were ended)XBase++XojoFOCUS4D QLInformix-4GLNATURALOpenEdge Advanced Business Language (OpenEdge ABL)OpenROAD (Ingres 4GL)RAMISSQLSpeedware EasyReporterLINCOracle ReportsOpenEdge Advanced Business Language (former Progress 4GL) Query/ResultsRPG-IIExtract data from files or database to create reports in a wide range of formats is done by the report generator tools.Ab InitioABAPAudit Command LanguageClarion Programming LanguageCorVisionCulpritADS/Online (plus transaction processing)EasytrieveFOCUSGraphTalkIDLIGOR ProInformix-4GLJSLLANSALabVIEWLiveCodeM-PowerMANTISMAPPER (Unisys/Sperry) now part of BISMARK-IV (Sterling/Informatics) now VISION:BUILDER of CASimulink a component of MATLABNATURALNomadOctavePL/SQLOpenEdge Advanced Business Language (former Progress 4GL)RRAMISSScilabSASSPSSSQL PLSSISSQRStataSynonWolfram LanguageXBase++XQueryXSLT4th Dimension (Software)LiveCodeMATLAB's GUIDEOmnis StudioOpenROADOpenEdge Advanced Business Language (former Progress 4GL) AppBuilderSuperTalkVisual DataFlexXojoXUL Can be used with XQueryAIMMSAMPLGAMSMathematicaMathProgMATLABC/ALGenexusSB+/SystemBuilderUnify VISIONSource:[12][13]1C:Enterprise programming languageAppceleratorAppianBizagiDronaHQ[14]LANSAM-PowerMicrosoft Power AutomateNode-REDOutSystemsPegasystemsPNMsoftOpenEdge Advanced Business Language (former Progress 4GL) ProgressServiceNowViziAppsOracle FormsOpenEdge Advanced Business Language (former Progress 4GL) ProVisionSB+/SystemBuilderActiveVFPCFMLLANSAOutSystemsWavemakerMaxMSPList of fourth-generation programming languagesDomain-specific languageRapid application development^ 35th Hawaii International Conference on System Sciences - 1002 Domain-Specific Languages for Software Engineering Archived May 16, 2011, at the Wayback Machine^ Arie van Deursen; Paul Klint; Joost Visser (1998). "Domain-Specific witness Languages:An Annotated Bibliography". Archived from the original on 2009-02-02. Retrieved 2009-03-15.^ MacDonell, Stephen (November 1993). "Software development, CASE tools and 4GLsA survey of New Zealand usage. Part 1". 750 New Zealand organisations. hdl:10523/928.^ "Roger Clarke's Software Generations".^ "Definition of fourth-generation language". PCMAG. Retrieved 7 May 2020.^ Martin, James. Application Development Without Programmers. Prentice-Hall, 1981. ISBN0-13-038943-9.^ "IBM Punch Cards". www.columbia.edu. Archived from the original on 2019-02-16. Retrieved 2019-02-03.^ "Data Mining Software, Data Analysis, and More: Unisys Business Information Server Features". Unisys. 2006-08-21. Archived from the original on 2006-08-21. Retrieved 2019-02-03.^ Louis Schlueter (1988). User-Designed Computing: The Next Generation. [book on report generator and MAPPER systems]^ Barbara Canning McNurlin; Ralph H. Sprague (2004). "Ch. 9". Information Systems Management in Practice (PDF). Pearson/Prentice Hall. ISBN978-0-13-101139-7.[permanent dead link]^ Forrest, Conner. "How Salesforce is using low-code orchestration to save 'floundering IoT projects'". ZDNet.^ Marvin, Rob (August 10, 2018). "The Best Low-Code Development Platforms for 2019". PCMAG.^ Sayer, Martin Heller and Peter (6 April 2018). "25 simple tools for building mobile apps fast". InfoWorld. Archived from the original on 18 November 2017. Retrieved 17 November 2017.^ "DronaHQ. Build apps without coding". www.dronahq.com. Archived from the original on 2019-08-09. Retrieved 2019-09-11.Retrieved from " If you work with databases, you know that the number of rows returned in a query can sometimes get overwhelming. In this article, were going to show you how to limit the number of rows returned in an Oracle database query so that you can work more efficiently and keep your data organized. The ROWNUM Pseudocolumn The most straightforward way to limit the number of rows returned in an Oracle database is to use the ROWNUM pseudocolumn. This pseudocolumn generates a unique number for each row returned in a query, and you can use it to limit the number of rows that are returned. Heres a simple example of how to use the ROWNUM pseudocolumn to return only the first 5 rows of a table: SELECT *FROM mytableWHERE ROWNUM