POSTGRES Version 4.0
Release Notes

1992/07/14

1. Introduction

These are the release notes for Version 4.0 of the POSTGRES database system from UC Berkeley. The database system and its installation procedure are covered in detail in the setup document for this release, which can be found in the file `$POSTGRESHOME/doc/postgres-setup.me`. Here, we cover only the most important differences from Version 3.1 and earlier versions of the system.

2. Aim

This release provides improved functionality over Version 3.1 in many areas of the system. In particular, significant advances have been made in POSTGRES functions and large objects.

3. Changes to POSTQUEL

3.1. define function

The syntax for defining functions has been changed slightly. POSTGRES now supports functions that can return sets of values. To declare such a function you use the keyword `setof` before the actual type name when specifying the return type:

```
define function hobbies
    (language = "postquel", returntype = setof hobbies_r)
    arg is (emp)
    as "retrieve (hobbies_r.all)
        where hobbies_r.person = $1.name"
```

The function `hobbies` is declared to return a set of instances from the class `hobbies_r`. Had `setof` been omitted, the `retrieve` would be executed exactly one time and the first instance retrieved would be the only return value. The primary reason for forcing this distinction is that POSTQUEL functions declared to return a single value can be used in a much more general fashion than functions returning a set of results.

POSTGRES does not support functions or operators with sets as arguments. For example:

```
define function alwaysone
    (language = "postquel", returntype = int4)
    as "retrieve (x = 1)"
```

Calls to `alwaysone` may be used as though they were constants. In contrast, if we had declared the return type to be `setof int4`, `alwaysone` would be forbidden as an argument to functions and/or operators.

POSTGRES now provides much better optimization of expensive functions. To aid the optimizer, the author of a C function can now choose to specify parameters which will be used to determine the function's "cost" in the query under consideration. For complete details see the Reference Manual page on functions.

3.2. NULL Detection

POSTGRES now recognizes the keywords `ISNULL` and `NOTNULL` and allows qualifications based on whether or not an attribute is null. The general syntax in qualifications is:

```
where expression ISNULL
where expression NOTNULL
```

3.3. Unified Syntax for Functions and Attributes

If a POSTQUEL function is declared that takes instances of a class as its only argument, it may be invoked in two different ways. For example, assume that the function `overpaid` has been declared in POSTQUEL to take an instance of the `emp` class and return a boolean value. Then the two queries

```
retrieve (emp.overpaid)
retrieve (overpaid = overpaid(emp))
```

are exactly equivalent. Similarly, attributes may be referenced in the same ways. If `emp` has an attribute `salary`, then

```
retrieve (emp.salary)
retrieve (salary = salary(emp))
```

are equivalent. Both of these may be considered parameterless method calls on instances of the `emp` class.

3.4. Binary Portals

It is now possible to get data out of POSTGRES in binary format. The query syntax is

```
retrieve iportal portal name (target list) qualifiers
```

Taking advantage of this feature should speed up `libpq` application programs considerably. See the Reference Manual for all of the details.

3.5. Arrays

In previous releases array indirection values were required to be integer constants. In Version 4.0 they can be any expression as long as the expression is of type int4. For example the following queries are now legal

```
retrieve (foo.a[1+2], foo[arrayind.i], foo.a)
retrieve (foo.a[arrayind.i]) where arrayind.i <= 3
```

4. General System Changes

4.1. Functions

The big story of the Version 4.0 release is improved POSTQUEL functions. POSTQUEL function support in previous releases suffered from the following limitations:

- They could take only a single argument, which had to be an instance of some class.
- They had to return a set of instances of some class.
- No run-time or declaration-time type checking was performed to verify that the arguments or return value of the function were correctly typed.

POSTQUEL functions in Version 3.1 of POSTGRES were implemented using the rewrite rules system. Version 4.0 supports query language functions as first-class citizens, and no longer relies on the rules system to implement them. As a result, the shortcomings listed above have been eliminated.

There are no special restrictions on the types or the number of arguments, nor is there any restriction on the type of the result. As outlined above, the syntax for defining a POSTQUEL function is analogous to defining a C function. The biggest difference is that the `as` clause must contain a quoted list of POSTQUEL queries. Within the body of the function, $*number* refers to the *number*th parameter passed in by the caller. See the Reference Manual for details.

C functions have also been improved. In previous releases C functions were restricted to taking at most one "tuple" argument, that is, an argument whose type is an instance of some class. In Version 4.0, C functions can take up to eight tuple arguments.[1]

Finally, POSTGRES provides enhanced optimization of both C and POSTQUEL functions. In the case of C functions, the optimizer takes user-specified parameters on cost into account when deciding how to execute a query.

POSTQUEL functions have one important limitation: system attributes (for example, `tmin`, `tmax`, and `oid`) of function return values and parameters cannot be referenced. The POSTGRES parser checks this condition when a function is defined and invoked, and prints a descriptive error message if the condition is violated. This limitation will be removed in a future release of the system.

4.2. Revised Support for Nested Relations

Version 3.1 of POSTGRES supported nested relations using tuple-valued attributes and rewrite rules. For example, given the declarations:

```
create person (name = char16, child = char16)

addattr (parent = person) to person

define rewrite rule find_parent
    on retrieve to person.parent do instead
      retrieve (person.all)
          where person.child = current.name
```

it was possible to refer to

```
person.parent.name
```

and have the correct name fetched by POSTGRES.

In Version 4.0 of POSTGRES, rewrite rules are no longer used to support nested relations. Instead, a function should be declared that returns tuples of the appropriate type. For example, in Version 4.0,

```
create person (name = char16, child = char16)

define function parent
```

[1] The 8-argument limit applies to **all** POSTGRES functions.

```
(language = "postquel", returntype = person)
arg is (person)
as "retrieve (person.all) where person.child = $1.name"
```

are equivalent to the rule and attribute additions shown above. The same nested-dot notation is supported for functions as was supported for rewrite rules.

As a side effect of the improved support for functions, it is not possible in Version 4.0 to create a relation which has an attribute that is of a complex (tuple) type. For example, the `addattr` command shown above is illegal in this release of POSTGRES, since the type of the attribute being added is `person`, and `person` is a class. Support for attributes of complex types will be reintroduced in a future release of POSTGRES.

4.3. Sets

POSTGRES now provides limited support for sets. Currently this comes via results of POSTQUEL functions (C functions should also be able to return a set of values; this will be available in a future release). The key design choice to point out is that when two set-valued POSTQUEL functions appear in the target list of a `retrieve` query, the result will be a set of "flattened" tuples. This set of tuples is the cross product of all the sets appearing the target list. For example, if you have two functions, `f` and `g`, where `f` returns 3 results and `g` returns 2, POSTGRES will return 6 tuples which make up all combinations of `f` and `g`. If either `f` or `g` retuns no results (i.e. the empty set) then we consider it to be a set of 1 element whose value is NULL. This choice was made primarily to avoid returning no results when a function in the target list returns nothing.

4.4. Large Objects and the Inversion File System

Support for large objects in POSTGRES has been improved. Version 3.1 supported large objects stored in UNIX files. User applications could open, seek, read, and write these large objects, but only inside dynamically-loaded functions. In addition, changes to large objects were not transaction-protected, and no support for historical access was available.

In Version 4.0, another large object implementation is provided. This implementation breaks large objects into "chunks," and stores each chunk in a tuple in the database. Updates are transaction-protected and time travel is available. Version 3.1 large objects ("UNIX large objects") are still supported. The new implementation ("Inversion large objects") is available by specifying the proper flags when an object is created. Details are provided in the Reference Manual.

In addition, a rudimentary file system has been built on top of the POSTGRES Inversion large object implementation. The Inversion file system provides transaction-protected access to user files, time travel on file system state, and the ability to do attribute- or content-based searches for file data.

At present, no kernel interface to Inversion is available, so the Inversion file system cannot be mounted and used in the same way as the native file system. Instead, a suite of library routines and utility programs give users access to the Inversion file system. Again, see the Reference Manual for details.

4.5. Indices on System Catalogs

POSTGRES now has secondary indices defined on the system catalogs. Initially, we have indexed `pg_proc`, `pg_attribute`, and `pg_type`. We expect that, after profiling

POSTGRES performance on large databases, additional indices will be added in future releases.

5. Miscellaneous

5.1. Big Bug Fixes

We have found and fixed a number of serious bugs with multi-user operation that were present in release 3.1. One bug sometimes caused a backend to read end-of-file from the socket on start up. Another much more destructive bug caused data from one database to be written to another database in certain circumstances. If the database being written to was `template1`, then the corruption would be passed on to whoever ran a subsequent `createdb`. Often `createdb` would just fail.

5.2. Reference Manual and User Manual

A serious effort was made to improve the dilapidated condition of both the Reference Manual and User Manual. The main focus was to root out all of the lies they contained, as well as documenting all of the new features. We will be paying much closer attention to bugs in these two documents. If you find something wrong in them, or something you can't understand, let us know about it (`bug-postgres@postgres.Berkeley.EDU`).

5.3. POSTGRES User Ids and Unix UIDs

The user ID of a POSTGRES registered user **must** match the user's UNIX user ID. In the new release, the user ID of the POSTGRES user in `/etc/passwd`is **not** presumed to be anything. We now adjust the system catalog entry for the user ID at installation time. It will happen automatically; there is no longer any need to edit `pg_user.h`.

5.4. Security

Security in POSTGRES is pretty much non-existent. We are working on providing it for a future release. For now the only check POSTGRES performs is at initialization time. The backend will lookup the front-end user to make sure s/he is a registered POSTGRES user before allowing the user to run any queries. Just so you are aware, it is not at all difficult for someone to circumvent this poor excuse for a lock-out mechanism. If you really care about your data, you should make backups of your `$POSTGRESHOME/data` directory regularly.

In previous releases, `pg_user` contained entries for each of the POSTGRES implementors. In Version 4.0, the only entry is for the user named `postgres`.

5.5. Append Returns an Object Id

When you run the POSTQUEL `append` command from `pqexec`, the command tag that you receive has the `oid` of the appended tuple in it. When you run `append` queries from the monitor you get output something like:

```
APPEND 206557
```

When two or more tuples are appended by the same command POSTGRES returns the invalid object ID zero.

6. Known Bugs

There are a few known bugs that we did not fix in the current release.

6.1. Indices and the Instance Level Rule System

The instance-level rule system essentially ignores indices, so if you are defining a rule on an indexed attribute, you should use the query rewrite rule system.

6.2. Retrieve Into and failed backends

If a backend fails while in the course of executing a `retrieve into` query, a spurious file, with the same name as the target class of the `retrieve into`, will be left in the database directory. This file can be safely deleted by the database administrator.

6.3. Known bugs list

A list of known bugs and suggested work-arounds can be anonymously FTP'd from `postgres.Berkeley.EDU` (128.32.149.1). This list is kept in the file `pub/postgres-v4r0.bugs`. We will make every attempt to keep this list up to date.

7. Machine-dependent Problems

7.1. SPARCstations running SunOS 4.0.3

POSTGRES has been known to crash SunOS 4.0.3 on SPARCstations, due to a SunOS bug in shared memory. This bug has not been observed on SunOS 4.1 and higher, so any reports of crashes on SunOS 4.1 and higher are especially appreciated.