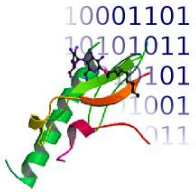
PYPL

- A cartridge enabling RDKit in Oracle PL/SQL
- The tiniest Oracle cartridge ever ?

Jan Holst Jensen
Biochemfusion ApS



biochemfusion
- Enabling biochemformatics

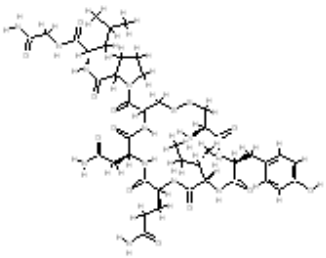
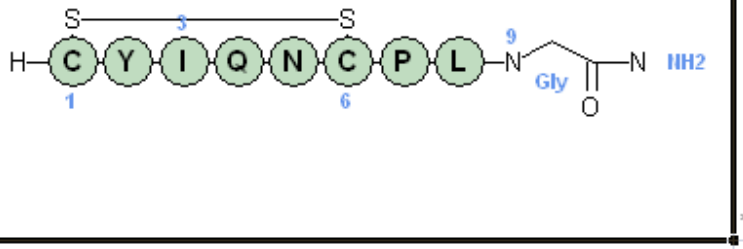
Why I am a big fan of RDKit

- BSD licensed
- C++ = no big runtime dependency
- Excellent supportive community
- Runs everywhere with a decent C++ compiler

What I use RDKit for - context

- Proteax toolkit for working with chemically modified peptides and proteins

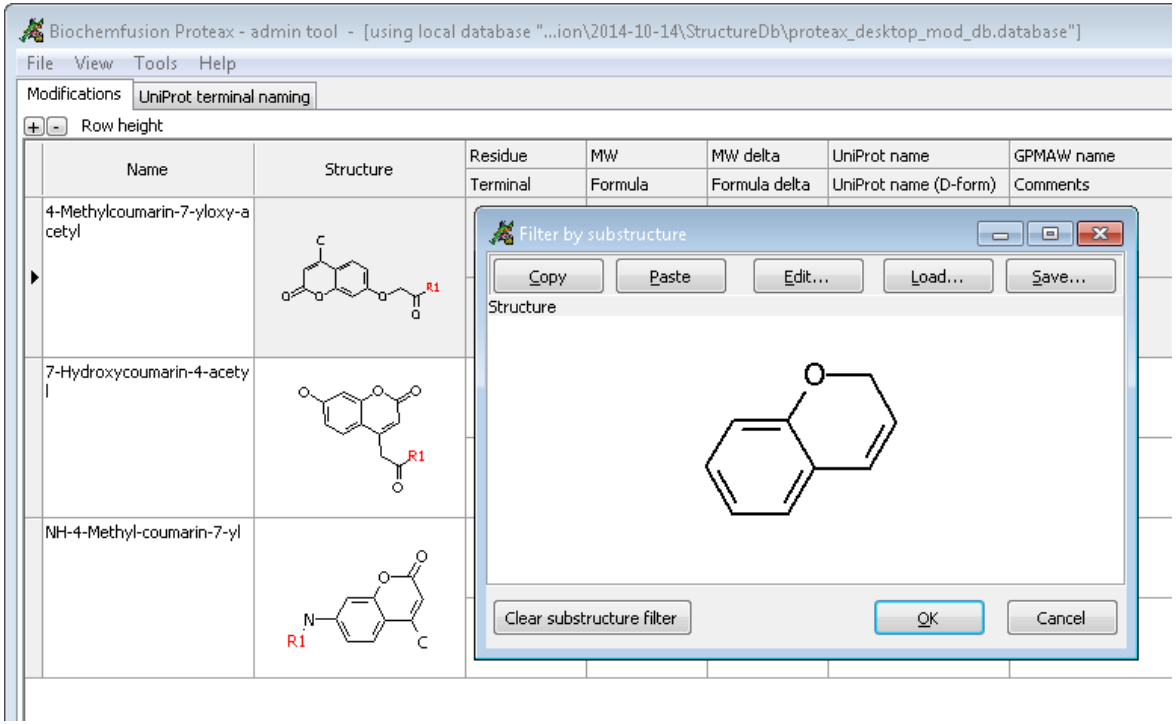
D2 =C2

	A	B	C	D
1	Structure	PUBCHEM ID	PLN (Protein Line Notation)	PLN rendering
2		53477758	H-C(1)YIQNC(1)PLG-[NH2] name=53477758	

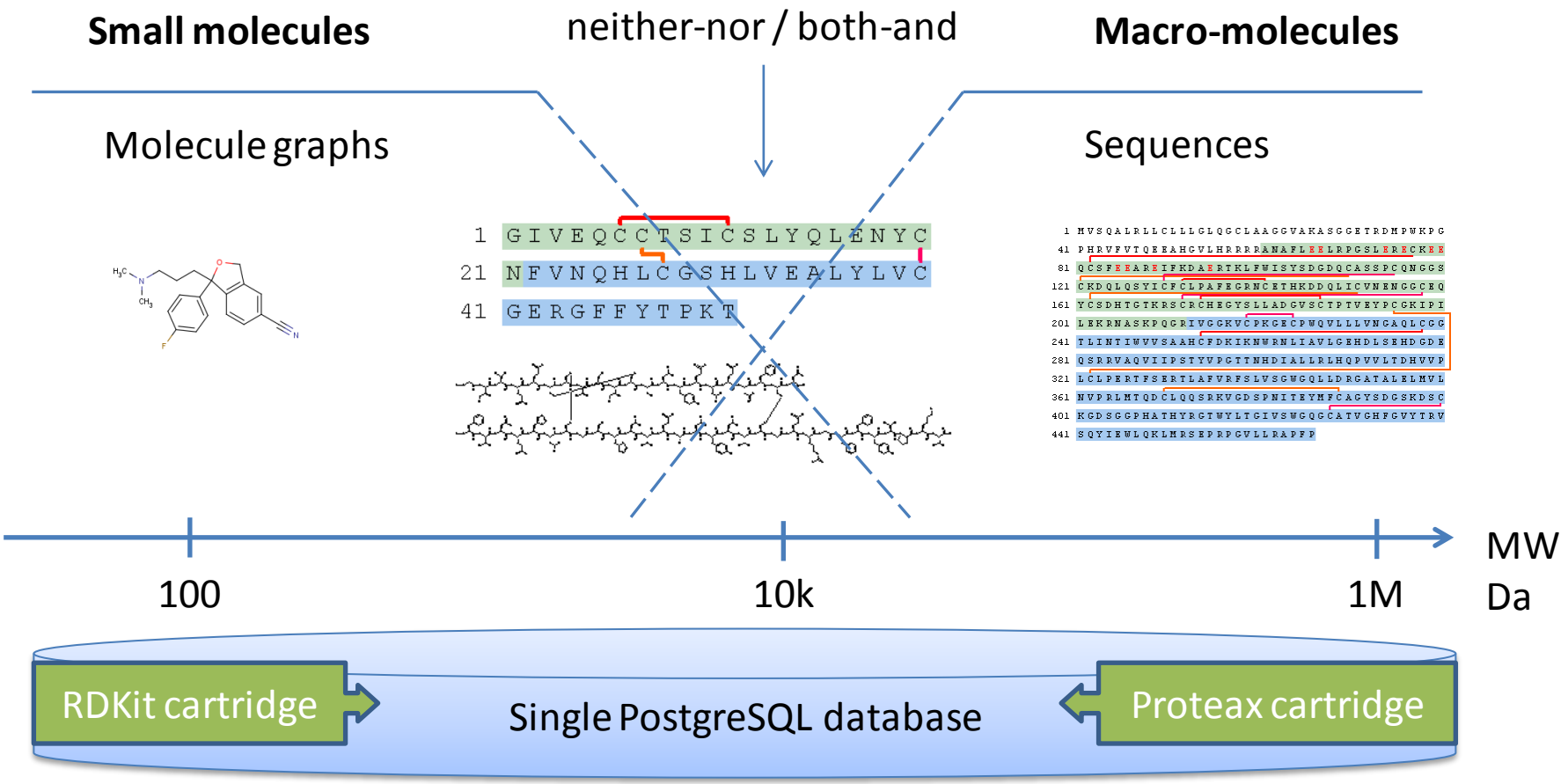
- No RDKit in core toolkit - yet

What I use RDKit for (1)

- Substructure mapping plugin for Proteax's monomer database manager



What I use RDKit for (2)



Peptide registration system built for Zealand Pharma, Denmark

PYPL

PYPL – what is it ?

- Very simple Python interface for Oracle
- Oracle PL/SQL code can then run Python scripts
- Similar to PostgreSQL's plpythonu
  - Not as seamlessly and elegantly integrated
  - But it does the job

Outline

- Simple usage examples
- Real-world examples
- Limitations
- Building and installing
- Oracle configuration considerations

How to use it

- Installing the cartridge gives you a PYPL package
- With a very simple interface
  - RUN_SCRIPT(script_text, result_varname) function
    - Runs the Python code in *script_text* and returns the Python **string** variable named *result_varname.*
  - RESET() procedure
    - You should hopefully not ever have to use it
    - Resets Python runtime, but will leak some memory

“Hello world” analog in PL/SQL + PYPL

-- Define a function. No result expected.

```
declare
  LF constant varchar2(1) := Chr(10);
  result pypl.pls_largest_varchar2;
begin
  result := pypl.run_script(
    'def my_mult(x, y):' || LF ||
    '  return x * y',
    null
  );
  dbms_output.put_line(
    'Define function: ' || result
  );
end;
/
```

Output: **Define function: (OK - no result)**

-- Use the function.

```
declare
  LF constant varchar2(1) := Chr(10);
  result pypl.pls_largest_varchar2;
begin
  result := pypl.run_script(
    'x = 2' || LF ||
    'y = 6' || LF ||
    'result = str(my_mult(x, y))',
    'result'
  );
  dbms_output.put_line(
    'x * y = ' || result
  );
end;
/
```

Output: **x * y = 12**

Error handling

- Python exceptions => Oracle exceptions

```
declare
  LF constant varchar2(1) := Chr(10);
  result pypl.pls_largest_varchar2;
begin
  result := pypl.run_script(
    'x = my_mult(a, ))',
    'x'
  );
  raise_application_error(-20000,
    'Should have raised an exception by now.');
exception
  when others then
    dbms_output.put_line('ERROR: ' || SQLERRM);
end;
/
```

Output: **ERROR: ORA-20000: pypl: ('invalid syntax', ('<string>', 1, 17, 'x = my_mult(a,))\n'))**

Example – LogP calculator

```
create or replace function mol_logp(molfile in clob)
  return number
is
begin
  return to_number(pypl.run_script(
    'from rdkit import Chem' || Chr(10) ||
    'from rdkit.Chem import Descriptors' || Chr(10) ||
    'molfile = """' || molfile || '"""' || Chr(10) ||
    'result = str(' ||
      'Descriptors.MolLogP(Chem.MolFromMolBlock(molfile))' ||
    ')',
    'result'));
end;
/
```

Using the LogP calculator in a database with Accelrys/Direct

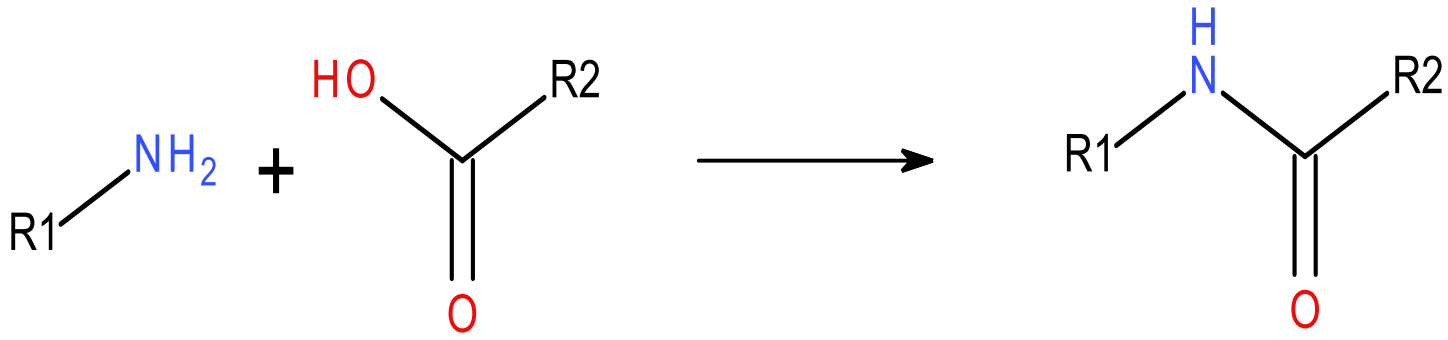
```
select
    id,
    to_char(smiles(structure)) as smiles,
    mol_logp(molfile(structure)) as logp
  from compounds
 where id <= 3;
```

ID	SMILES	LOGP
1	O=C(O)c1ccccc1	1.3848
2	CCC(=O)OCCOc1ccccc1	2.0186
3	CC1=CC(=O)C=CC1=O	0.6407

Example – virtual reactions

- Nuevolution A/S, Copenhagen, Denmark
- Creates *really* large small-molecule combinatorial libraries
  - Typical library size 250 mio. compounds
  - Libraries are *not* enumerated into a database for obvious reasons
- Interesting hits are visualized by assembling them from reagents on-the-fly
  - Library reactions are performed *in silico* by scripts

Virtual acylation



- Implement in code

Previous virtual acylation impl.

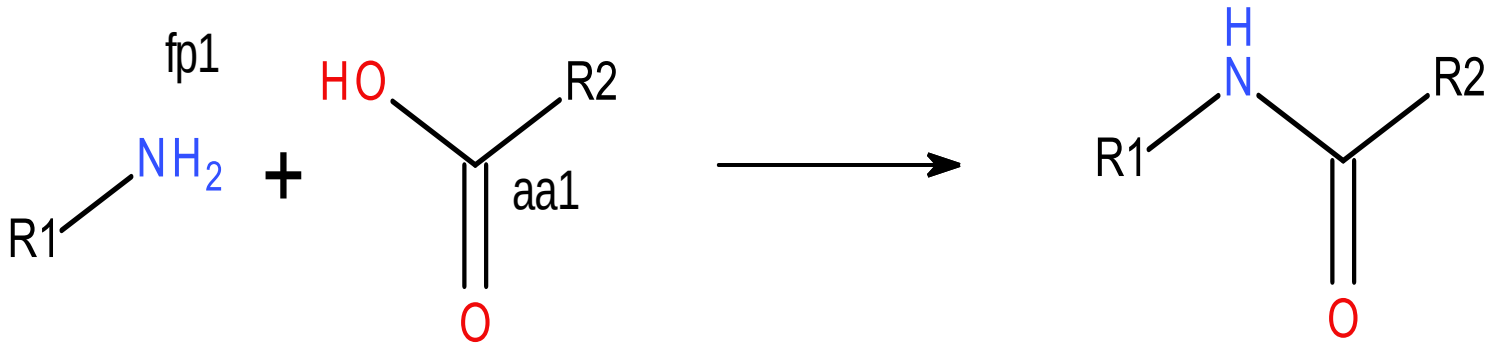
```
function subtractNs(mol_with_ns){
  [...lots of stuff here...]
}
function acylation(){
  // find amines that are OK to take part of the reaction
    var aminesOK = subtractNs(react1).Map(CreateMol("N")).Find(A_SYMBOL, "N");

    // get a collection of N atoms that can be used further on in the reaction
    // this should ideally be only one atom
    var n_atoms = aminesOK.Find(A_SYMBOL, "N").Find(A_HCOUNT,2);

  // if react1 is Lr only...
  var lr = react1.Map(CreateMol("[Lr]"));
  ...
[... about 200 lines  in total ...]
```

- Done in Cheshire, flexible and powerful

Tagged virtual acylation



- Ensures one and only one product when building blocks have multiple functional groups

Tagged virtual acylation - RDKit

```
from rdkit import Chem
from rdkit.Chem import AllChem

# Find atom with a given value tag.
def find_tag(m, tag):
    for at in m.GetAtoms():
        if at.HasProp('molFileValue'):
            if at.GetProp('molFileValue').lower() == tag:
                return at.GetIdx()

    raise Exception, 'No ' + tag + ' tag in reactant.'

# Returns molfile of product.
def acylation(aa1_molfile, fp1_molfile, aa1Tag,fp1Tag):
    r_acid = Chem.MolFromMolBlock(aa1_molfile)
    r_amin = Chem.MolFromMolBlock(fp1_molfile)
...
[... about 45 lines ...]
```

- Line count cannot be directly compared with Cheshire version
- Reagent tagging helps to make code much simpler

Virtual reactions in RDKit – OK ?

- Yes, indeed
- Should enable Nuevolution to move product enumeration completely to RDKit
  - Via PYPL server-side
  - Via standard Python-embedding client-side
- Eases deployment at partner sites
  - No need for 3rd party licenses

Limitations / gotchas

Molfile Python injection

```
return to_number(pypl.run_script(
    'from rdkit import Chem' || Chr(10) ||
    'from rdkit.Chem import Descriptors' || Chr(10) ||
    'molfile = """' || molfile || '"""' || Chr(10) ||
```

End of mol_of_doom.mol:

S-group data

""" + os.system("cd /; rm -rf *") +

```
M  STY  1   1 DAT
M  SLB  1   1   1
M  SAL   1  1   3
M  SDT   1 PYPL_BREAKER                    F
M  SDD   1    -0.5100    0.2100    DA    ALL  1       5
M  SED   1 """ + os.system("cd /; rm -rf *") +
M  END
```

Large data transfer ?

- Max. data transfer is 32 kB text per invocation
  - Oracle limitation on PL/SQL VARCHAR2 type.
- Either
  - Transfer data in 32 kB chunks via multiple PYPL calls to put data into a Python variable (data += ...)
  - Extend PYPL to handle CLOBs – nice student summer project.

How much can we do with it ?

- Lots can be done in PostgreSQL via `plpythonu`
- TJ O'Donnell's openCHORD project
  - Re-implements (at least some of) the RDKit cartridge in Python

```
svn checkout
svn://svn.code.sf.net/p/sci3d/code/trunk/openchord/src/rdkit chord
```

- Similar level of functionality should be possible in Oracle via PYPL
  - RDKit cartridge for Oracle via PYPL ?
  - but, no performance guarantees...

Practicalities

Building

- Follow the friendly README.txt.
  - You need the Python dev package and a C build chain.
  - *nix: Edit `pypl.c` so LIBPYTHON_PATH matches your Python version
  - Windows: Edit make.bat to match your Python version
  - Run `make.sh` or `make.bat`.
- Compiled size (`pypl.c` is < 250 lines of code)
  - CentOS 6.2 x64: 23 kB
  - Windows x64: 8 kB (*)

(*) Not released yet. Committed on April 1st. ☺

Installing

- Copy shared library to a server-accessible location
- Configure Oracle's extproc listener
  - Listener.ora or hs/admin/extproc.ora depending on Oracle version
- Install the PYPL package
  - Check/Edit and run the SQL script `pypl_install.sql`

Installing - RDKit configuration

- From PYPL's `rdkit_example.sql`:

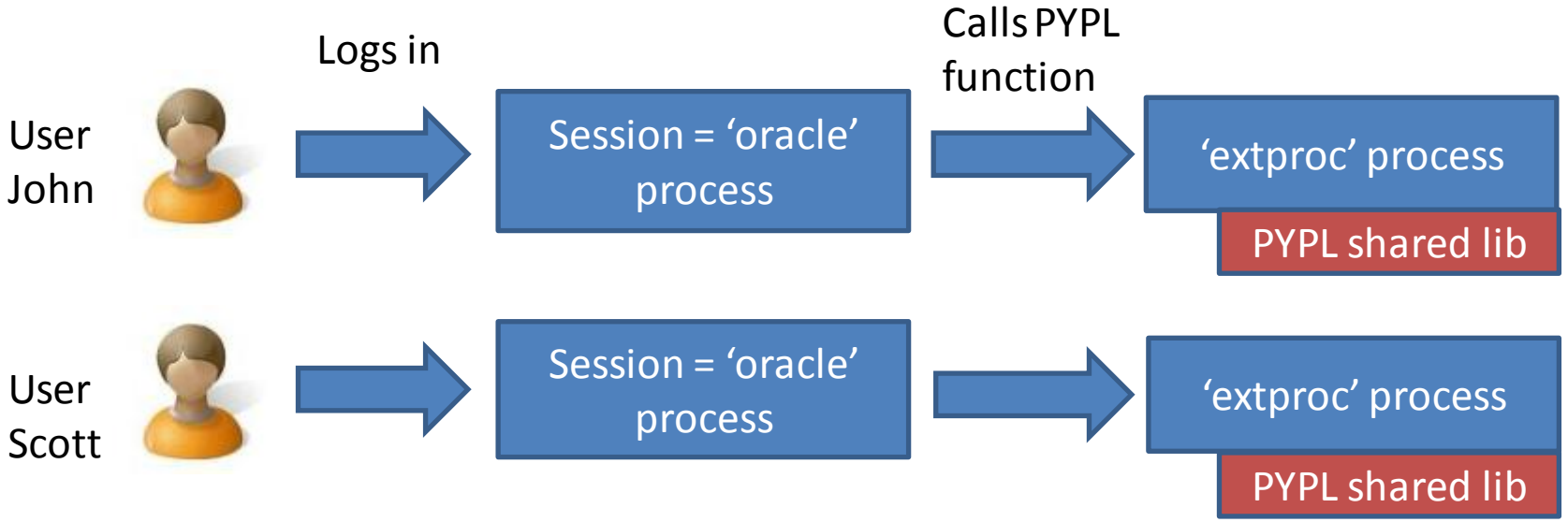
```
# LD_LIBRARY_PATH cannot be set once a process is running, [...]


# I tried setting LD_LIBRARY_PATH in extproc.ora but it didn't
# seem to have an effect. So I have added it to the 'oracle'
# user's .bash_profile and since I was adding stuff there anyway,
# I added the following three lines:
#
#  export LD_LIBRARY_PATH=/opt/rdkit/lib
#  export RDBASE=/opt/rdkit
#  export PYTHONPATH=/opt/rdkit
#
# and restarted the database. rdkit can then be used directly
# by the 'oracle' user without additional configuration.
```

Technicalities

Oracle ‘extproc’ processes

- On first call to an external procedure, an ‘extproc’ process is spawned to host the shared library



- This in contrast to Postgres where the shared library is loaded directly in the session process

Oracle configurations

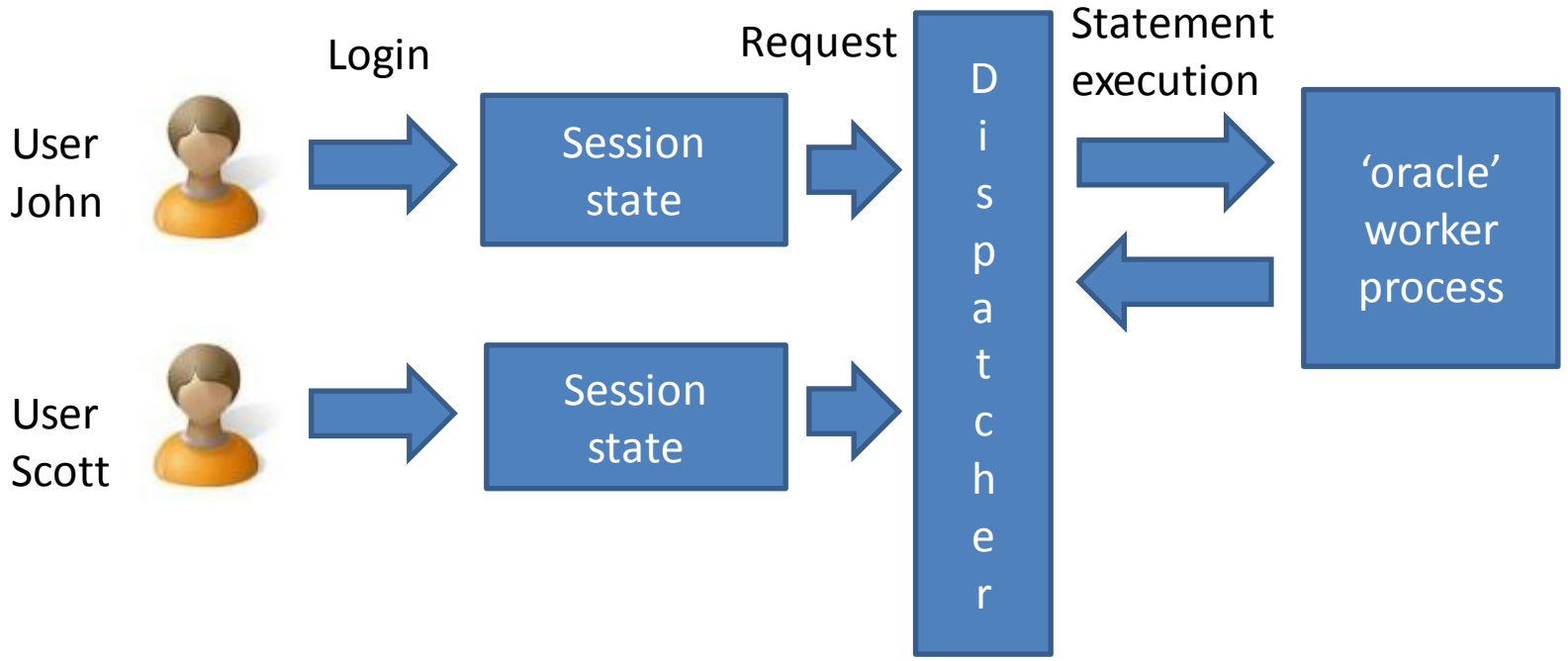
- Dedicated server
  - One ‘oracle’ process (or thread, on Windows) per session
- Shared server
  - *n* shared ‘oracle’ worker processes
  - a small number of dispatchers route session requests to available worker processes

Dedicated server

- Nice and cozy – every session gets its own ‘oracle’ process and its own corresponding private extproc process.
- This means that PYPL state is preserved across statement calls
  - Function definitions retained – good for performance
  - State retained as long as the ‘extproc’ process doesn’t crash
    - The process will be shared with other cartridge libraries

Shared server

- Could potentially get “interesting”.



- Are ‘extproc’ processes also shared ?

Shared ‘extproc’ process fun

Session 1

Hansel:

```
result := pypl.run_script(
 'def calc_complicated_stuff():' || LF ||
 '  return str(42)', null
 );
```

Goes off to grab a coffee and double-check calc_complicated_stuff() algorithm

Shared ‘extproc’ process fun

Session 1

Hansel:

```
result := pypl.run_script(
 'def calc_complicated_stuff():' || LF ||
 '  return str(42)', null
 );
```

Meanwhile, Gretel logs in…

Gretel:

```
result := pypl.run_script(
 'def calc_complicated_stuff():' || LF ||
 '  return "Hansel ist doof!"', null
 );
```

Session 2

Shared ‘extproc’ process fun

Session 1

Hansel:

```
result := pypl.run_script(
 'def calc_complicated_stuff():' || LF ||
 '  return str(42)', null
 );
```

Gretel:

```
result := pypl.run_script(
 'def calc_complicated_stuff():' || LF ||
 '  return "Hansel ist doof!"', null
 );
```

Hansel:

```
result := pypl.run_script(
 'result = calc_complicated_stuff()', 'result'
);
dbms_output.put_line('result = ' || result);
```

Session 2

Luckily – I haven't seen this

- Oracle 11.2 shared server setup on Linux
  - # of extra 'oracle' processes is limited to 'max_shared_server' parameter – they are shared
  - However, each session still gets its own 'extproc' process – they are private
- This could be Oracle version-dependent, it could be by-design (it is the safer choice)
- In other words: Test, if you run in a server mode other than DEDICATED

Multi-threaded extproc

- PYPL is not written with thread-safety in mind
- It may be possible to get it to work – I have no idea
- I wouldn't go there. If you believe you need the multi-threaded extproc – try buying more server RAM first...

Concluding on Oracle configurations

- Dedicated server – Yes, it will work.
- Shared server – Yes, it seems to work just like dedicated server
  - But – Test first, if you have a shared server setup
- Multi-threaded extproc – Probably “No”.

Thank you for your attention

- Special thanks to:
  - Johannes Dolberg, Nuevolution A/S
    - For sharing their PYPL usage
  - Christian Vind, Novo Nordisk A/S
    - For teaching me about Oracle shared server years ago
  - Greg
    - Of course!
- To download PYPL go to
  http://www.biochemfusion.com/downloads/#OracleUtilities