Συστήματα Διαχείρισης Βάσεων Δεδομένων
Εργαστηριακή Διάλεξη
PostgreSQL – part I

Τμήμα Πληροφορικής, Πανεπιστήμιο Πειραιώς,

Data Science Lab. (datastories.org)

Ανδρέας Τριτσαρώλης
andrewt@unipi.gr

Outline

- PostgreSQL (basic features)
- *Hands on (Queries)
- Indexing
- Planner

Outline

- PostgreSQL (basic features)
- *Hands on (Queries)
- Indexing
- Planner

PostgreSQL ~ Features & Extensions

- **Features**
  - complex queries
  - foreign keys
  - triggers
  - views
  - transactional integrity
  - full-text searching
  - limited data replication

- **Extensions**
  - new data types
  - functions (aggregate)
  - operators
  - index methods

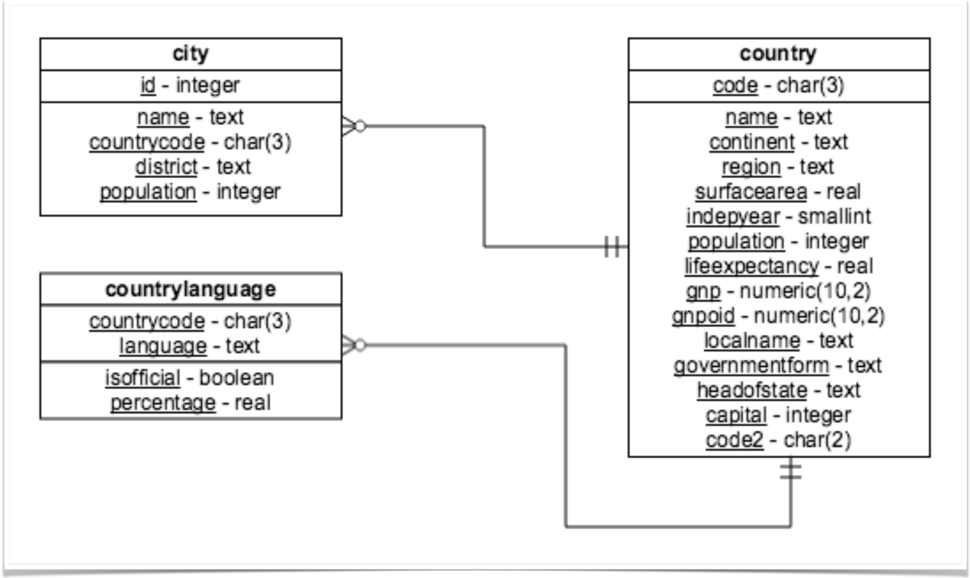
PostgreSQL ~ Basics

create	database | schema | table (as) | type | view
drop	table | view | type | index
alter	add | drop | add constraint | rename to | alter column | modify
insert	into
copy	from | to
rename	table | column | etc.
set operations	union | intersect | except
string operations (pattern matching)	like
aggregate functions	avg | min | max | sum | count
order by	asc | desc
grouping	having
nested queries	set membership | set comparison | etc.
joins	cross join | qualified joins (inner / outer)

Outline

- PostgreSQL (basic features)
- *Hands on (Queries)
- Indexing
- Planner

DB Schema “Cities | Countries | Languages”



Hands on ~ Examples

1. **Derived Relations —** Sum countries' population, where the first letter is 'A'

2. **Derived Relations —** Sum countries' population and percentage (An/A), where the first letter is 'A' or 'An'

3. **Join —** Cities & countries (name), belonging to 'Asia' & city's population is higher that 4000000 people

4. **Join —** Cities & countries (name), belonging to 'Asia' & (optionally) city's population is higher that 4000000 people

Hands on ~ Examples

5. **Join –** Country (code), city (name) & language, having percentage greater than 50% & population is higher than 4000000 people

6. **Join –** Country (name) and language (percentage, official) having the maximum percentage and the language is official

7. **Join** – Country & city (names) as well as language, sorted by country's code

Hands on ~ Answers

Hands on ~ Answers

1. **Derived Relations** - - Sum population (countries), where the first letter is ‘A’

```
select total_a
from
        (select sum(population) as total_a
         from Country
         where name like 'A%') as t1;
```

Hands on ~ Answers

2. **Derived Relations** - - Sum population (countries), percentage (An/A), where the first letter is ‘A’ or ‘An’

```
select total_an, total_a, (cast (total_an as real) / cast (total_a as real)) as percentage
from
        (select sum(population) as total_a
        from Country
        where name like 'A%') as t1,
        (select sum(population) as total_an
        from Country
        where name like 'An%') as t2
```

Hands on ~ Answers

3. **Join** - - Cities & Countries (name), belonging to ‘Asia’ & city’s population is higher that 4000000 people

```
select City.name, Country.name
from City inner join Country on City.CountryCode = Country.Code
where Country.Continent ='Asia' and City.Population > 4000000;
```

Hands on ~ Answers

4. **Join** - - Country & City (name), belonging to ‘Asia’ & (optionally) city’s population is higher that 4000000 people

```
select t1.name, t2.name
from
(select name, code
from country
where continent = 'Asia') as t1 left outer join
(select name, countrycode
from city
where population > 4000000) as t2 on t2.countrycode = t1.code;
```

Hands on ~ Answers

5. **Join** - - Country (code), city (name) & language, having percentage greater than 50% & population is higher than 4000000 people

```
select *
from
(select language, countrycode
from countrylanguage
where percentage > 50) as t1 natural join
(select name, countrycode
from city
where population > 4000000) as t2
```

Hands on ~ Answers

6. **Join** - - Country (name), countrylanguage (percentage, official) having the maximum percentage and the language is official

```
select *
from (
    select countrycode, max(percentage) as max_pct
    from countrylanguage
    where isofficial=true
    group by countrycode
) as cl_pct_max
 inner join countrylanguage on cl_pct_max.countrycode = countrylanguage.countrycode
where countrylanguage.percentage = cl_pct_max.max_pct;
```

Hands on ~ Answers

7. **Join** - - Join - - Country & city (names) as well as language, sorted by country's code

```
select country.name, city.name, countrylanguage.language
from city inner join countrylanguage on city.countrycode = countrylanguage.countrycode
inner join country on city.countrycode = country.code
order by city.countrycode;
```

Outline

- PostgreSQL (basic features)
- *Hands on (Queries)
- Indexing
- Planner

Indexing ~ Introduction

- **Indices**
    - An index allows the database server to find and retrieve specific rows much faster than it could do without an index.
    - A PostgreSQL index is a data structure that provides a **dynamic mapping** from search predicates to sequences of tuple IDs from a particular table.
    - The returned tuples are intended to match the search predicate, although in some cases the predicate must be rechecked on the actual tuples, since the index may return a superset of matching tuples.
    - PostgreSQL supports **several types of indices** that target different categories of workloads (i.e., B-tree, Hash, GiST, GIN).

⚠ Enhance DB performance… but add overhead to the DB system (they should be used sensibly)

Indexing ~ Introduction (cont.)

```
SELECT value
FROM t2
WHERE num = 1;
```

- With no advance preparation, the system would have to scan the entire T2 table to find all matching entries…
- **Index on the num column**: it can use a more efficient method for locating matching rows (a few levels deep into a search tree)

```
CREATE INDEX T2_id_index ON T2 (num);
```

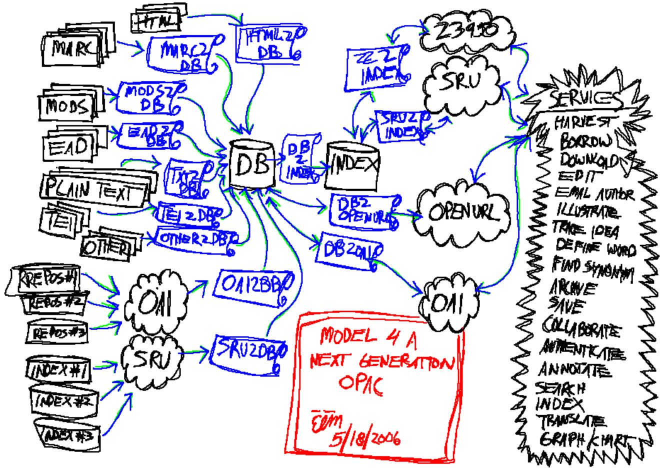
- Once an index is created, no further intervention is required (the system will update the index when the table is modified)
- Indices can be added to & removed from tables at any time

Removing an index: DROP INDEX

Indexing ~ Introduction (cont.)

- **Join searches**: An index defined on a column that is part of a join condition can significantly speed up queries with joins!

- After an index is created, the system has to **keep it synchronized with the table**
    - Adds overhead to data manipulation operations
    - Indices that are seldom or never used in queries should be removed

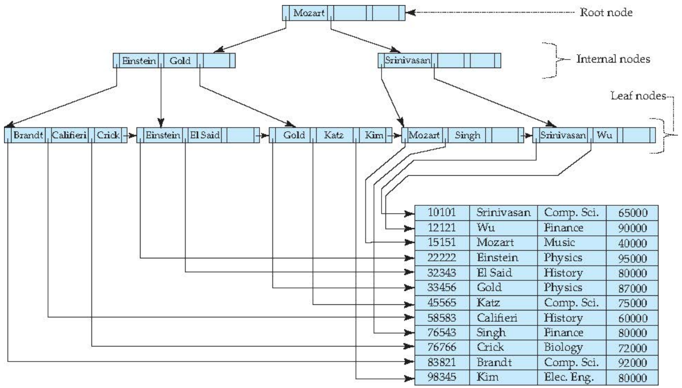


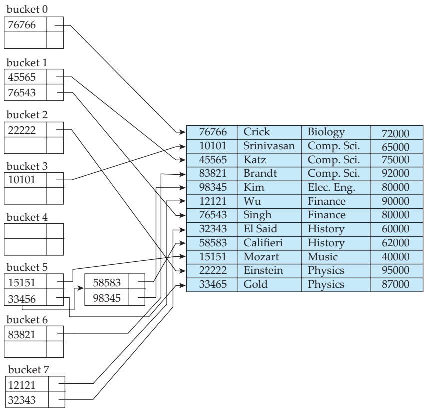
Creating an index on a large table can take a long time …

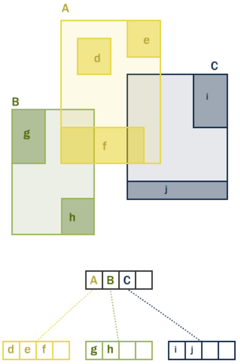
Indexing ~ Index Types

- **B-tree** (default - fit the most common situations)
- Hash
- **Generalized Search Tree (GiST)** (R-tree like)
- **Generalized Inverted Index (GIN)**

⚠ Each index type uses a different algorithm that is best suited to **different types of queries**





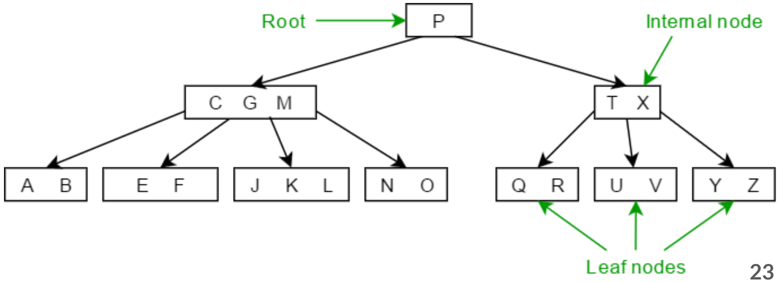


Indexing ~ B-Trees

- The **B-tree** is the default index type.
- B-trees can efficiently support **equality** and **range queries** on sortable data, as also certain **pattern-matching operations** such as some cases of like expressions.

- **PostgreSQL query planner**: will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators

<	<=	=	>=	>



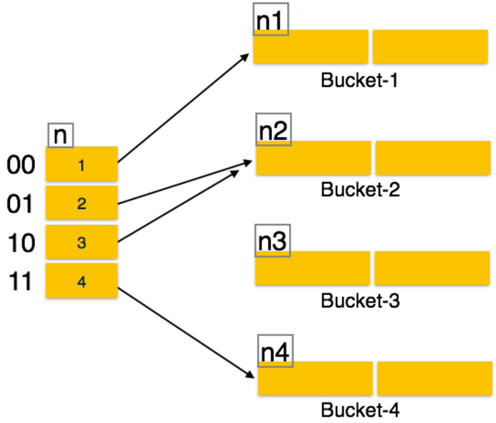
Indexing ~ B-Trees (cont.)

- **BETWEEN** & **IN** can also be implemented with a B-tree index search
- **IS NULL** or **IS NOT NULL** condition on an index column can be used with a B-tree index
- **LIKE**: The optimizer can also use a B-tree index for queries involving the pattern matching operator

- B-tree indices can also be used to retrieve data in sorted order

⚠ This is not always faster than a simple scan and sort, but it is often helpful!

Indexing ~ Hash



- PostgreSQL's hash indices are an implementation of **linear hashing**.
- Useful only for simple **equality operations**
- **PostgreSQL query planner**: will consider using a hash index whenever an indexed column is involved in a comparison using the '=' operator

```
CREATE INDEX name ON table USING HASH (column);
```

⚠ The hash indices used by PostgreSQL had been shown to have lookup performance no better than that of B-trees while having considerably larger size and maintenance costs.

✓ In PostgreSQL 10 & 11, the hash index implementation has been significantly improved: hash indices now support write-ahead logging, can be replicated, and performance has improved as well.

Indexing ~ Generalized Search Tree (GiST)

- Extensible indexing structure supported by PostgreSQL.
- An infrastructure within which many **different indexing strategies** can be implemented (not a single kind of index).
- The GiST index is based on a balanced tree-structure similar to a B-tree.
- The standard distribution of PostgreSQL includes GiST operator classes for several **two-dimensional geometric data types**, which support indexed queries using these operators

`<<` `&<` `&>` `>>` `<<|` `&<|` `|&>`

`|>>` `~=` `&&` `@>` `<@`

Indexing ~ Generalized Search Tree (GiST) (cont.)

- GiST indices are also capable of optimizing **nearest-neighbor searches**

Example

✓ Find the ten places closest to a given target point

```
SELECT *
FROM places
ORDER BY location <-> point '(101,456)' LIMIT 10;
```

Indexing ~ Generalized Inverted Index (GIN)

- The GIN index is designed for speeding up queries on **multi-valued elements**, such as text documents, JSON structures and arrays.
- Provides extensibility by allowing an index implementor to specify custom “strategies” for specific data types.
- The standard distribution of PostgreSQL includes GIN operator classes for **one-dimensional arrays**, which support indexed queries using these operators

<@	@>	=	&&

Indexing ~ Multicolumn Indices

```
CREATE TABLE test1

(major int,

minor int,

name varchar

);
```

```
SELECT name

FROM test1

WHERE major = constant AND minor = constant;
```

```
CREATE INDEX test1_mm_idx ON test1 (major, minor);
```

Indexing ~ Multicolumn Indices (cont.)

- An index can be defined on **more than one column** of a table
- **B-tree**, **GiST**, **GIN** index types support multicolumn indices (up to 32 columns can be specified!)

⚠ Multicolumn indices should be used sparingly. In most situations, an index on a single column is sufficient & saves space and time

Indexing ~ Order by & planner

- An index may be able to deliver the rows to be returned by a query in a specific **sorted order** (B-tree)
- The planner will consider satisfying an **ORDER BY** specification
  - by scanning an available index that matches the specification
  - by scanning the table in physical order & doing an explicit sort

⚠ For a query that requires scanning a large fraction of the table, an explicit sort is likely to be faster than using an index because it requires less disk I/O due to following a sequential access pattern!

Indexing ~ Order by & planner (cont.)

- Indices are more useful when **only a few rows need be fetched**
- You can adjust the ordering of a B-tree index by including the options **ASC**, **DESC**

```
CREATE INDEX ON T1 (num DESC);
```

Indexing ~ Indices on Expressions

- An index column need not be just a column of the underlying table, but can be a **function** or **scalar expression** computed from one or more columns of the table
- This feature is useful to obtain fast access to tables based on the results of computations

The index expressions are not recomputed during an indexed search, since they are already stored in the index

Indexing ~ Indices on Expressions (cont.)

```
SELECT *
FROM T1
WHERE lower(name) = 'value';
```

```
CREATE INDEX t1_lower_col1_idx ON T1 (lower(name));
```

✓ We can also combine columns

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Indexing ~ Partial Indices

- **Partial index**: an index built over a subset of a table
  - One major reason for using a partial index is to **avoid indexing common values**
  - Since a query **searching for a common value** (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all

This reduces the size of the index

- ✓ It will **speed up those queries that do use the index**
- ✓ It will also speed up many table update operations because the **index does not need to be updated in all cases**

Indexing ~ Partial Indices (cont.)

Example

- Suppose you are storing web server access logs in a database …
    - Most accesses originate from the IP address range of your organization but some are from elsewhere (e.g., employees on dial-up connections)
    - If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet

Indexing ~ Partial Indices (cont.)

```
CREATE TABLE access_log (

url varchar,

client_ip inet,

...

);
```

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)

WHERE NOT (client_ip > inet '192.168.100.0' AND

client_ip < inet '192.168.100.255');
```

Indexing ~ Partial Indices (cont.)

- A typical query that can use this index is

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

- A query that cannot use this index is:

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

Outline

- PostgreSQL (basic features)
- *Hands on (Queries)
- Indexing
- Planner

Planner ~ Introduction

- The task of the planner/optimizer is to **create an optimal execution plan**.
- A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results.
- If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately **selecting the execution plan that is expected to run the fastest**.
- **Selects** and (especially) **Joins** can often be time consuming, hence the planner will always review many query plans and pick the one that it finds to be the most efficient.

Planner ~ Example

- Let's run a simple SELECT query on our IMDb Database[1] and analyze the query plan that the planner picked.

```
explain analyze select * from name_basics where deathyear = '2016';
```

- Our objective is to view all the information available for every person that died during 2016.

- By using "explain analyze", instead of the standard output (the table containing the actual information), Postgres will return the detailed **Query plan** for the aforementioned query, including an **estimation of the cost** (in an arbitrary form) and the **Execution** and **Planning times**.

[1] https://www.imdb.com/interfaces/

Planner ~ Example

- Let's take a look at the Query plan

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1000.00..150306.46 rows=4855 width=61) (actual time=1.035..1466.625 rows=4475 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on name_basics (cost=0.00..148820.96 rows=2023 width=61) (actual time=0.435..1441.471 rows=1492 loops=3)
5	Filter: (deathyear = '2016'::text)
6	Rows Removed by Filter: 3033248
7	Planning Time: 0.255 ms
8	Execution Time: 1467.531 ms

- The selected plan scans the table sequentially and filters for 'deathYear' = 2016. The complete arbitrary cost value for the Sequential Scans is aprox. 150K and the Execution time is 1467 ms.

Planner ~ Example

- Let's create a B-Tree index on the column 'deathYear' and reevaluate the query plan for the same query.

```
create index idx_name_basics_deathyear on name_basics using btree("deathyear");
```

- The index that we named 'idx_name_basics_deathyear' took more than 15 seconds to complete and needs 195 MB of disk space.

```
select pg_size_pretty(pg_relation_size('idx_name_basics_deathyear'));
```

	pg_size_pretty text
1	195 MB

Planner ~ Example

- Now let's evaluate the query plan for the same query as before.

	QUERY PLAN text
1	Bitmap Heap Scan on name_basics (cost=109.25..19023.30 rows=5782 width=67) (actual time=1.824..8.576 rows=4612 loops=1)
2	Recheck Cond: ("deathYear" = 2016)
3	Heap Blocks: exact=4337
4	-> Bitmap Index Scan on "ix_name_basics_deathYear" (cost=0.00..107.80 rows=5782 width=0) (actual time=1.111..1.111 rows=4612 lo...
5	Index Cond: ("deathYear" = 2016)
6	Planning time: 0.284 ms
7	Execution time: 8.963 ms

- The time gained is obvious just by looking at the **Execution time.** This query plan consist of two steps, the **Bitmap Index Scan** that locates all the Heap Blocks that satisfy the condition 'deathYear' = 2016 using the index and the **Bitmap Heap Scan** that fetches and outputs the specified Blocks.

Planner ~ Example

- This time let's evaluate a query that consists of two conditions separated with **OR**.

```
explain analyze select * from name_basics where deathyear < '2016' or deathyear > '1955';
```

	QUERY PLAN text
1	Bitmap Heap Scan on name_basics (cost=170145.28..407431.51 rows=8944692 width=61) (actual time=10570.193..17114.978 rows=8966095 loops=1)
2	Recheck Cond: ((deathyear > '2016'::text) OR (deathyear < '1955'::text))
3	Rows Removed by Index Recheck: 112305
4	Heap Blocks: exact=35330 lossy=66079
5	-> BitmapOr (cost=170145.28..170145.28 rows=8962675 width=0) (actual time=10558.222..10558.223 rows=0 loops=1)
6	-> Bitmap Index Scan on idx_name_basics_deathyear (cost=0.00..165331.21 rows=8944370 width=0) (actual time=10537.435..10537.436 rows=8946880 loops=1)
7	Index Cond: (deathyear > '2016'::text)
8	-> Bitmap Index Scan on idx_name_basics_deathyear (cost=0.00..341.72 rows=18305 width=0) (actual time=20.779..20.779 rows=19215 loops=1)
9	Index Cond: (deathyear < '1955'::text)
10	Planning Time: 0.177 ms
11	Execution Time: 17683.926 ms

Planner ~ Example

- Let’s run a new query.

```
explain analyze select * from name_basics where "deathyear" - "birthyear" > 100;
```

- The query plan reminds us that the database was scanned Sequentially, something that we should -by now- know is not efficient.

	QUERY PLAN text
1	Seq Scan on name_basics (cost=0.00..260964.05 rows=3212223 width=67) (actual time=0.227..1536.531 rows=825 loops=1)
2	Filter: (("deathYear" - "birthYear") > 100)
3	Rows Removed by Filter: 9635845
4	Planning time: 0.133 ms
5	Execution time: 1536.782 ms

Planner ~ Example

- Let's create a useful index. Many people may search our database using the age of a person. This is not a column that we have available. We can -of course- create it, but do we need it in order to efficiently return the needed information? I.e. can we create an index on a column that does not exist?

```
create index idx_name_basics_age on name_basics using btree (("deathyear" - "birthyear"));
```

	QUERY PLAN text
1	Bitmap Heap Scan on name_basics (cost=60131.16..224728.51 rows=3212223 width=67) (actual time=0.232..3.910 rows=825 loops=1)
2	Recheck Cond: (("deathYear" - "birthYear") > 100)
3	Heap Blocks: exact=813
4	-> Bitmap Index Scan on ix_name_basics_age (cost=0.00..59328.11 rows=3212223 width=0) (actual time=0.149..0.149 rows=825 loop...
5	Index Cond: (("deathYear" - "birthYear") > 100)
6	Planning time: 0.271 ms
7	Execution time: 3.994 ms

Planner ~ Example

- Always remember:
    - Indexes are good when they are used a lot and put food on the table.
    - Indexes are not good when they sit around in disk space that we pay for by the hour…
    - Indexes do not have to be created on a column. They can instead be created on an expression, like “deathYear” - “birthYear”.

```
SET enable_seqscan = off;
```

Should NEVER be used!

References

- Abraham Silberschatz, Henry F. Korth, S. Sudarshan (2020), DATABASE SYSTEM CONCEPTS, SEVENTH EDITION. McGraw-Hill Education, 2 Penn Plaza, New York.
- Abraham Silberschatz, Henry F. Korth, S. Sudarshan (2015), Συστήματα Βάσεων Δεδομένων, Γ’ Έκδοση. Εκδόσεις Μόσχος Γκιούρδας.
- The PostgreSQL Global Development Group, PostgreSQL 9.6rc1 Documentation. 2016.
- PostGIS 2.3.3dev Manual, SVN Revision (15388).