**http://stat-computing.org/dataexpo/2009/the-data.html**

## The data

[The data](http://stat-computing.org/dataexpo/2009/the-data.html) consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12 gigabytes when uncompressed. To make sure that you're not overwhelmed by the size of the data, we've provide two brief introductions to some useful tools: [linux command line tools](http://stat-computing.org/dataexpo/2009/unix-tools.html) and [sqlite](http://stat-computing.org/dataexpo/2009/sqlite.html), a simple sql database.

## The challenge

The aim of the data expo is to provide a **graphical** summary of important features of the data set. This is intentionally vague in order to allow different entries to focus on different aspects of the data, but here are a few ideas to get you started:

* When is the best time of day/day of week/time of year to fly to minimise delays?
* Do older planes suffer more delays?
* How does the number of people flying between different locations change over time?
* How well does weather predict plane delays?
* Can you detect cascading failures as delays in one airport create delays in others? Are there critical links in the system?

You are also welcome to work with interesting subsets: you might want to compare flight patterns before and after 9/11, or between the pair of cities that you fly between most often, or all flights to and from a major airport like Chicago (ORD). Smaller subsets may also help you to match up the data to [other interesting datasets](http://stat-computing.org/dataexpo/2009/supplemental-data.html).

**Variable descriptions**

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| 1 | Year | 2008 |
| 2 | Month | 1-12 |
| 3 | DayofMonth | 1-31 |
| 4 | DayOfWeek | 1 (Monday) - 7 (Sunday) |
| 5 | DepTime | actual departure time (local, hhmm) |
| 6 | CRSDepTime | scheduled departure time (local, hhmm) |
| 7 | ArrTime | actual arrival time (local, hhmm) |
| 8 | CRSArrTime | scheduled arrival time (local, hhmm) |
| 9 | UniqueCarrier | [unique carrier code](http://stat-computing.org/dataexpo/2009/supplemental-data.html) |
| 10 | FlightNum | flight number |
| 11 | TailNum | plane tail number |
| 12 | ActualElapsedTime | in minutes |
| 13 | CRSElapsedTime | in minutes |
| 14 | AirTime | in minutes |
| 15 | ArrDelay | arrival delay, in minutes |
| 16 | DepDelay | departure delay, in minutes |
| 17 | Origin | origin [IATA airport code](http://stat-computing.org/dataexpo/2009/supplemental-data.html) |
| 18 | Dest | destination [IATA airport code](http://stat-computing.org/dataexpo/2009/supplemental-data.html) |
| 19 | Distance | in miles |
| 20 | TaxiIn | taxi in time, in minutes |
| 21 | TaxiOut | taxi out time in minutes |
| 22 | Cancelled | was the flight cancelled? |
| 23 | CancellationCode | reason for cancellation (A = carrier, B = weather, C = NAS, D = security) |
| 24 | Diverted | 1 = yes, 0 = no |
| 25 | CarrierDelay | in minutes |
| 26 | WeatherDelay | in minutes |
| 27 | NASDelay | in minutes |
| 28 | SecurityDelay | in minutes |
| 29 | LateAircraftDelay | in minutes |

# Using a database

For moderate initial investment in time, and a large investment in space (>30 gigabytes), you can considerably speed up access to the data by loading it into a database. This page shows you how to do so for [sqlite](http://sqlite.org), an open-source sql database.

## Create database

Creating a new database couldn't be simpler: just run the following on the command line to create a new database in the current directory.

sqlite3 ontime.sqlite3

## Create table and import data

Next create a table with fields that match the csv files:

create table ontime (

Year int,

Month int,

DayofMonth int,

DayOfWeek int,

DepTime int,

CRSDepTime int,

ArrTime int,

CRSArrTime int,

UniqueCarrier varchar(5),

FlightNum int,

TailNum varchar(8),

ActualElapsedTime int,

CRSElapsedTime int,

AirTime int,

ArrDelay int,

DepDelay int,

Origin varchar(3),

Dest varchar(3),

Distance int,

TaxiIn int,

TaxiOut int,

Cancelled int,

CancellationCode varchar(1),

Diverted varchar(1),

CarrierDelay int,

WeatherDelay int,

NASDelay int,

SecurityDelay int,

LateAircraftDelay int

);

Then load the data with the .import directive:

.separator ,

.import 2008.csv ontime

.import 2007.csv ontime

.import 2006.csv ontime

Unfortunately this also imports the column headers, so remove them with this sql:

delete from ontime where typeof(year) == "text";

## Adding indices

To speed up access to the data, you'll also want to add indices. The code below adds a few that I've found useful, but you'll want to think about your needs and add your own. (It's most efficient to create the indices after all the data has been loaded, so make sure you've done that first.)

create index year on ontime(year);

create index date on ontime(year, month, dayofmonth);

create index origin on ontime(origin);

create index dest on ontime(dest);

**Stackoverflow**

**http://stackoverflow.com/questions/2887878/importing-a-csv-file-into-a-sqlite3-database-table-using-python**

**1**

import csv, sqlite3

conn = sqlite3.connect("pcfc.sl3")

curs = conn.cursor()

curs.execute("CREATE TABLE PCFC (id INTEGER PRIMARY KEY, type INTEGER, term TEXT, definition TEXT);")

reader = csv.reader(open(**'PC.txt', 'r'), delimiter='|'**)

for row in reader:

to\_db = [unicode(row[0], "utf8"), unicode(row[1], "utf8"), unicode(row[2], "utf8")]

curs.execute("INSERT INTO PCFC (type, term, definition) VALUES (?, ?, ?);", to\_db)

conn.commit()

**2**

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

import sys, csv, sqlite3

def main():

con = sqlite3.connect(sys.argv[1]) # database file input

cur = con.cursor()

cur.executescript("""

DROP TABLE IF EXISTS t;

CREATE TABLE t (COL1 TEXT, COL2 TEXT);

""") # checks to see if table exists and makes a fresh table.

with open(sys.argv[2], "rb") as f: # CSV file input

reader = csv.reader(f, delimiter=',') # no header information with delimiter

for row in reader:

to\_db = [unicode(row[0], "utf8"), unicode(row[1], "utf8")] # Appends data from CSV file representing and handling of text

cur.execute("INSERT INTO neto (COL1, COL2) VALUES(?, ?);", to\_db)

con.commit()

con.close() # closes connection to database

**3**

import csv, sqlite3

import logging

**def \_get\_col\_datatypes(fin):**

dr = csv.DictReader(fin) # comma is default delimiter

fieldTypes = {}

for entry in dr:

feildslLeft = [f for f in dr.fieldnames if f not in fieldTypes.keys()]

if not feildslLeft: break # We're done

for field in feildslLeft:

data = entry[field]

# Need data to decide

if len(data) == 0:

continue

if data.isdigit():

fieldTypes[field] = "INTEGER"

else:

fieldTypes[field] = "TEXT"

**# TODO: Currently there's no support for DATE in sqllite**

if len(feildslLeft) > 0:

raise Exception("Failed to find all the columns data types - Maybe some are empty?")

return fieldTypes

**def escapingGenerator(f):**

for line in f:

yield line.encode("ascii", "xmlcharrefreplace").decode("ascii")

def csvToDb(csvFile, outputToFile = False):

# TODO: implement output to file

with open(csvFile,mode='r', encoding="ISO-8859-1") as fin:

dt = \_get\_col\_datatypes(fin)

fin.seek(0)

reader = csv.DictReader(fin)

# Keep the order of the columns name just as in the CSV

fields = reader.fieldnames

cols = []

# Set field and type

for f in fields:

cols.append("%s %s" % (f, dt[f]))

# Generate create table statement:

stmt = "CREATE TABLE ads (%s)" % ",".join(cols)

con = sqlite3.connect(":memory:")

cur = con.cursor()

cur.execute(stmt)

fin.seek(0)

reader = csv.reader(escapingGenerator(fin))

# Generate insert statement:

stmt = "INSERT INTO ads VALUES(%s);" % ','.join('?' \* len(cols))

cur.executemany(stmt, reader)

con.commit()

return con

**4**

I am "converting" a large (~1.6GB) CSV file and inserting specific fields of the CSV into a SQLite database. Essentially my code looks like:

import csv, sqlite3

conn = sqlite3.connect( "path/to/file.db" )

conn.text\_factory = str #bugger 8-bit bytestrings

cur = conn.cur()

cur.execute('CREATE TABLE IF NOT EXISTS mytable (field2 VARCHAR, field4 VARCHAR)')

reader = csv.reader(open(filecsv.txt, "rb"))

for field1, field2, field3, field4, field5 in reader:

cur.execute('INSERT OR IGNORE INTO mytable (field2, field4) VALUES (?,?)', (field2, field4))

Everything works as I expect it to with the exception... IT TAKES AN INCREDIBLE AMOUNT OF TIME TO PROCESS. Am I coding it incorrectly? Is there a better way to achieve a higher performance and accomplish what I'm needing (simply convert a few fields of a CSV into SQLite table)?

\*\*EDIT -- I tried directly importing the csv into sqlite as suggested but it turns out my file has commas in fields (e.g. "My title, comma"). That's creating errors with the import. It appears there are too many of those occurrences to manually edit the file...

any other thoughts??\*\*

python csv sqlite3

shareimprove this question

It's a big file. How long does it take? – Blender May 9 '11 at 20:58

How many duplicate records are there? If there are a lot, it would probably be faster to keep a local set of records that have already been inserted, and skip the call to the SQL entirely for the duplicates.

Here are some MySQL bulk load speed tips

What operating system and Python version are you using

"It appears there are too many of those occurrences to manually edit the file..". Let's think. Too many to change manually? If only you had a programming language that would allow you to write a program to reformat a CSV file into a TAB-delimited file. Any ideas what language could be used to write a program like that?

It's possible to import the CSV directly:

sqlite> .separator ","

sqlite> .import filecsv.txt mytable

http://www.sqlite.org/cvstrac/wiki?p=ImportingFiles

+1: And. It still may take an incredibly long time to process.

Doesn't seem like there's a built-in way of escaping by default. Also, the quotes will be literals within the string. It might make sense to change the text using a CSV parse and outputting with a different separator but that might defeat the purpose of using the import tool in the first place.

Try: .mode csv instead of .separator, see: stackoverflow.com/questions/14947916/import-csv-to-sqlite/… – NumesSanguis Dec 3 '14 at 14:40

Chris is right - use transactions; divide the data into chunks and then store it.

"... Unless already in a transaction, each SQL statement has a new transaction started for it. This is very expensive, since it requires reopening, writing to, and closing the journal file for each statement. This can be avoided by wrapping sequences of SQL statements with BEGIN TRANSACTION; and END TRANSACTION; statements. This speedup is also obtained for statements which don't alter the database." - Source: http://web.utk.edu/~jplyon/sqlite/SQLite\_optimization\_FAQ.html

"... there is another trick you can use to speed up SQLite: transactions. Whenever you have to do multiple database writes, put them inside a transaction. Instead of writing to (and locking) the file each and every time a write query is issued, the write will only happen once when the transaction completes." - Source: How Scalable is SQLite?

import csv, sqlite3, time

def chunks(data, rows=10000):

""" Divides the data into 10000 rows each """

for i in xrange(0, len(data), rows):

yield data[i:i+rows]

if \_\_name\_\_ == "\_\_main\_\_":

t = time.time()

conn = sqlite3.connect( "path/to/file.db" )

conn.text\_factory = str #bugger 8-bit bytestrings

cur = conn.cur()

cur.execute('CREATE TABLE IF NOT EXISTS mytable (field2 VARCHAR, field4 VARCHAR)')

csvData = csv.reader(open(filecsv.txt, "rb"))

divData = chunks(csvData) # divide into 10000 rows each

for chunk in divData:

cur.execute('BEGIN TRANSACTION')

for field1, field2, field3, field4, field5 in chunk:

cur.execute('INSERT OR IGNORE INTO mytable (field2, field4) VALUES (?,?)', (field2, field4))

cur.execute('COMMIT')

print "\n Time Taken: %.3f sec" % (time.time()-t)

Another user following this code ran into a problem trying to use len() with their CSV reader: stackoverflow.com/questions/18062694/… – rutter Aug 5 '13 at 16:39

As it's been said (Chris and Sam), transactions do improve a lot insert performance.

Please, let me recommend another option, to use a suite of Python utilities to work with CSV, csvkit.

To install:

pip install csvkit

To solve your problem

csvsql --db sqlite:///path/to/file.db --insert --table mytable filecsv.txt

Try using transactions.

begin

insert 50,000 rows

commit

That will commit data periodically rather than once per row.

## Sqlite and R

If you use R, be sure to check out RSQlite. This lets you easily get the data out of sqlite into R data frames:

install.packages("RSQLite")

library(RSQLite)

ontime <- dbConnect("SQLite", dbname = "ontime.sqlite3")

from\_db <- function(sql) {

dbGetQuery(ontime, sql)

}

from\_db("select count(\*), tailnum from ontime group by tailnum")

tails <- from\_db("select distinct tailnum from ontime")

==================================================================================

Many fields such as diversions or cancellations have blank values. mongoimport would include a lot of these blank field values. To avoid field sparsity, we used [an upload Python script](http://t.umblr.com/redirect?z=https%3A%2F%2Fgithub.com%2F10gen-interns%2Fbig-data-exploration%2Fblob%2Fmaster%2FBasic-Flights%2Fsrc%2FinputMongo.py&t=NWY4MjUyZTUxNzZhODZhNzE1YjgxOWQ0NWU1NmQyODA0ZmVmZTMyOCxrRTdmSEFGdQ%3D%3D) to weed out the unnecessary fields and incomplete entries. This means that all fields which are blank or zero will not be included.

import os

import datetime

import time

import pprint

import csv

import glob

import sys

from pymongo import MongoClient

portNum = 27017

flightsDir = ""

# Get the environment variables for the port number and the directory for the flights csv file

try:

portNum = int(os.environ["PORT"])

flightDir = os.environ["FLIGHTS"]

except KeyError:

print "Please set the environment variables $PORT and $FLIGHTS."

sys.exit(1)

# Start a connection to the mongod instance

client = MongoClient("localhost", portNum)

coll = client["flying"]["flights"]

# ALL fields

fields = ["year", "quarter", "month", "dayOfMonth", "dayOfWeek"

, "date", "carrierId", "airlineId", "carrier", "tailNum"

, "flightNum", "origAirportId", "origCityId", "origAirport"

, "origCity", "origStateId", "origState", "origWAC", "destAirportId"

, "destCityId", "destAirport", "destCity", "destStateId", "destState"

, "destWAC", "crsDepTime", "depTime", "depDelay", "taxiOut", "wheelsOff"

, "wheelsOn", "taxiIn", "crsArrTime", "arrTime", "arrDelay","cancelled"

, "cancelCode", "diverted", "crsElapsedTime", "elapsedTime"

, "airTime", "numFlights", "distance", "carrierDelay"

, "weatherDelay", "nasDelay", "securityDelay", "lateAircraftDelay"

, "numDivAirportLandings", "divReachedDest", "divElapsedTime"

, "divArrDelay", "divDistance"]

# integer fields

integerFields = ["year", "quarter", "month", "dayOfMonth"

, "dayOfWeek", "airlineId", "flightNum", "origAirportId"

, "origCityId", "origWAC", "destAirportId", "destCityId", "destWAC"

, "depDelay", "taxiOut", "taxiIn","arrDelay", "crsElapsedTime"

, "elapsedTime", "airTime", "numFlights", "distance"

, "carrierDelay", "weatherDelay", "nasDelay", "securityDelay"

, "lateAircraftDelay", "numDivAirportLandings"

, "divElapsedTime", "divArrDelay", "divDistance"]

# boolean fields

booleanFields = ["diverted", "divReachedDest"]

# time fields

timeFields = ["crsDepTime", "depTime", "wheelsOff"

, "wheelsOn", "crsArrTime", "arrTime"]

correspondingTimeFields = {"crsDepTime": "crsArrTime"

, "depTime" : "arrTime"

, "wheelsOff": "wheelsOn"}

# fields that should have indices on them to speed up computations

indexFields = ["depDelay", "arrDelay", "tailNum", "origStateId"

, "depStateId"]

Takes in a doc, and corrects all the dates that are initially incorrect.

This includes arriving the next day and or delays causing next day flights

"""

def correctDays(doc):

# Some flights actually leave and or arrive the day or days after

# their schedule.

if "crsDepTime" in doc and "depDelay" in doc:

flewDate = doc["crsDepTime"] + datetime.timedelta(minutes=doc["depDelay"])

# if the scheduled and actual times are on different days

if flewDate.day != doc["crsDepTime"].day:

doc["depTime"] = flewDate

if "crsArrTime" in doc and "arrDelay" in doc:

arrDate = doc["crsArrTime"] + datetime.timedelta(minutes=doc["arrDelay"])

if arrDate.day != doc["crsArrTime"].day:

doc["arrTime"] = arrDate

# If the arrival time is before the departure time, this means that

# the plane actually arrived the next day, need to change manually

for attr, value in correspondingTimeFields.items():

if attr in doc and value in doc and doc[attr] > doc[value]:

doc[value] = doc[value] + datetime.timedelta(days=1)

return doc

lsToDoc -

takes in a line from the csv and "prepares" a new doc to be inserted

into a MongoDB collection

"""

def lsToDoc(ls):

doc = dict(zip(fields, ls))

# first parse the date, docs missing the date is useless

if "date" not in doc:

return {}

date = time.strptime(doc["date"], "%Y-%m-%d")

doc["date"] = datetime.datetime(date.tm\_year, date.tm\_mon, date.tm\_mday)

date = doc["date"]

# delete "cancelled", it becomes 0 in cancelCode

del doc["cancelled"]

for attr, value in doc.items():

# delete any empty columns

if not isinstance(value, datetime.datetime) and len(value) == 0:

del doc[attr]

elif attr in integerFields:

doc[attr] = int(float(value))

elif attr in booleanFields:

doc[attr] = bool(value)

elif attr in timeFields:

# All time fields aside from date of flight comes in the form

# hrmn where hour is in 1 to 24 and min is 00 to 59

hour, mins = int(value[:len(value)-2]), int(value[len(value)-2:])

if hour == 24:

hour = 0

# this is the next day

date = date + datetime.timedelta(days=1)

doc[attr] = datetime.datetime(date.year, date.month, date.day, \

hour, mins)

elif attr == "cancelCode":

# cancellation codes: A, B, C, D. Set as binary

# mapping { A:1, B:2, C:3, D:4 }

doc[attr] = ord(value)-64

doc = correctDays(doc)

return doc

Given the port number, flights directory, and mongo collection client

import the data in the files into mongoDB using bulk insertion

"""

def importFiles():

# Iterate through the files in the Flights directory

for fname in glob.glob(flightsDir + "/\*.csv"):

print "Importing file ", fname

with open(fname, "r") as f:

# skip the first line in the file, it's a column header line

f.readline()

# bulkData is used for bulk insertion to speed up the insertion time

bulkData = []

reader = csv.reader(f, delimiter=",", quotechar="\"")

numDone = 0

for line in reader:

# Turn a line into a MongoDB document

doc = lsToDoc(line)

# Discard lines that aren't useful if they don't have date

if len(doc) > 0:

bulkData.append(doc)

# Insert the data once the list reaches 1,000

if len(bulkData) % 1000 == 0:

numDone += 1000

coll.insert(bulkData)

if numDone % 100000 == 0:

print str(numDone) + " documents have been inserted so far."

bulkData = []

# Add the straggler data

coll.insert(bulkData)

def createIndices():

for index in indexFields:

print "Creating index for ", index

coll.create\_index(index)

if \_\_name\_\_ == "\_\_main\_\_":

importFiles()

print "====FINISHED IMPORTING===="

createIndices()

print "====FINISHED CREATING INDICES==== "

We wrote a script, [masterNNumbers.py](http://t.umblr.com/redirect?z=https%3A%2F%2Fgithub.com%2F10gen-interns%2Fbig-data-exploration%2Fblob%2Fmaster%2FBasic-Flights%2Fsrc%2FmasterNNumbers.py&t=MDhlZmZmMjI3OGZhNjk0ZDM1YjIzZTYwODQ0ZTI5NmE2OGM5OWU0ZixrRTdmSEFGdQ%3D%3D), with the function addAge that adds an **age** field into the documents for which a corresponding **tailNum** (also known as the n-number) exists in the current **flights** collection in the **flying** database. Since a single aircraft can have different departure and arrival times, an aircraft with a specific tail number must be updated in several documents. We used the [multi flag](http://t.umblr.com/redirect?z=http%3A%2F%2Fdocs.mongodb.org%2Fmanual%2Fcore%2Fupdate%2F%23update-multiple-documents&t=Y2UwYmI0ZWU2OTI2YjE0OGY3YzEwNzUzNDU2OTIzOTQxYTQ3MTU2YixrRTdmSEFGdQ%3D%3D) to enable multiple updates per aircraft.

import os

import datetime

import time

import pprint

import csv

import glob

import sys

from pymongo import MongoClient

portNum = 27017

flightsDir = ""

try:

# Get the environment variables for the port number and the directory for the flights csv file

portNum = int(os.environ['PORT'])

flightsDir = os.environ["FLIGHTS"]

except KeyError:

print "Please set the environment variables $PORT and $FLIGHTS"

sys.exit(1)

# Start a connection to the mongod instance

client = MongoClient("localhost", portNum)

coll = client["flying"]["flights"]

"""

takes in a line in the list and 'prepares' a new doc to be inserted into the MongoDB

Collection

"""

def setAircraftAge(ls, mongoCollClient):

# fields to include

# note: "n-number" is assumed to start with an N

fields = ["nNumber", "yearMfr"]

fieldcols = [0, 4]

# Only get the fields that we want

newls = [ls[i].strip() for i in fieldcols]

doc = dict(zip(fields, newls))

# yearMfr should be stored as an integer

if len(doc["yearMfr"]) > 0:

doc["yearMfr"] = int(doc["yearMfr"])

# calculate age of flights

age = 2013 - doc["yearMfr"]

# store in flights collection in flying database

mongoCollClient.update( { "tailNum": "N"+doc["nNumber"] }

, {"$set": {"age" : age}}

, multi= True)

"""

Add the Age of flights according to the MASTER directory

"""

def addAge(flightsDir, mongoCollClient):

with open(flightDir + "/MASTER.txt", "r") as f:

# ifnore header

f.readline()

reader = csv.reader(f, delimiter=",")

read = 0

for line in reader:

setAircraftAge(line, mongoCollClient)

read += 1

if read % 1000 == 0:

print read

if \_\_name\_\_ == "\_\_main\_\_":

addAge(flightsDir, coll);

print "====FINISHED ADDING AGE===="