autofit Documentation

Release 1

Renzo Tiranti

Jun 03, 2019

CONTENTS:

CHAPTER

ONE

AIM AND BASIC SPECIFICATIONS OF THE AUTOFIT SYSTEM

By Autofit we mean an IT system, built around Python and a relation database aimed at the full automation of calibration tasks for IPV purposes. Though at least initially focused on Equity, efforts will be spent to maintain the implementation as general as possible.

The basic issues in implementing such system are:

1. IPV methodologies are interdependent and related through a hierachical dependency tree - for example, volatility calibration is performed only after the forward and implied volatility calculation
2. Different methodologies can be applied to the calibration of the same parameters depending on the available market data
3. New methodoogies can be inserted in the hierarchy tree with minimal changes to the code
4. The calibration tasks must be run in parallel and ideally on different machine
5. The system must cope with exeptions and failures of particular market data with no disruption on unrelated tasks
6. In case of catastrophic collapse, the system can be restarted from the last valid state

CHAPTER

TWO

THE MRP DATABASE, SLOTS AND PRODUCERS

The system is based on two abstractions of data and processing: *slots* and *producers*. The slots are linked to entries in a database called MRP table, where the acronyms stands for Market Data, Reports and Parameters.

2.1 Slot

The slot in an abstraction of a collection of homogeneous data, which is indexed by a string *name* and whose data is indicated by a date and a collection of supplementary keys. It is called slot because once data is generated and wrapped in a slot, it is inserted into the MRP table. Example of slot's name may be **MarketData.Vanilla.Totem** or **ImpliedVolatility.Vanilla.Totem** or **ProductRate.TRS.Totem**.

Todo: It is unclear how the slot can be defined, i.e. it represents a homogeneous dataset, but how is the scope determined? For example, option prices should be grouped into a unique slot based on expiry/strike/underlying or just underlying? Probably the grouping should be also determined by the primary keys, so the question is: what are the primary keys for a slot?

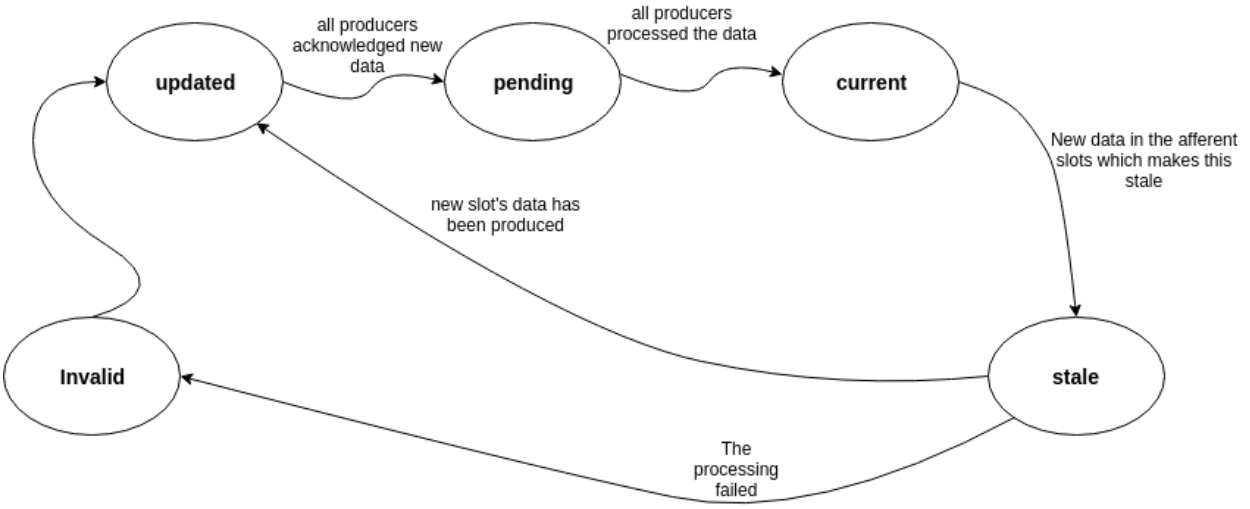
This must be decided or it won't be possible to handle proxy.

Slots are characterized by a **state**

1. **updated** The slot's data was updated and processing has not started yet. We need This state to **trigger** Producers, for example when the system is restarted
2. **pending** The slot's data has just been generated and all linked producers are processing the data
3. **current** The slot's data has been processed by all the linked producers.
4. **invalid** Some non-recoverable error happened during the processing
5. **stale** At least one of the upstream slots changed status to Failed/Updated/Stale
6. **void** This is the starting state of a slot, when no data has been produced yet

Warning: The slots' state is a mess, we need clarity, and a clear definition of what those states really mean!!!

The chart below shows how the state evolves and what triggers changes in the state



2.1.1 Brainstorming

How do we handle proxies, or cases in which many input slots feed into a fewer numbers of output slots?

To to this there must be a logic that performs aggregation of slots' keys.

Warning: Some slots may need to be linked, for example product rate and dividends are related.

2.1.2 The SID

Each data in the slot is identified by a unique id the *SID*. This is not a duplicate of the slot's key as the date triggers a new SID.

2.1.3 The Listeners

To each slot is associated a list of producers, or *listeners* in this context, that has the slot as required or optional, see next section for the definition. This list is required so that, upon generation of new or updated slots, the depending producers are notified.

2.1.4 A journey inside the slot life!

- A slot is produced and it goes into the table with 'updated' state.
- All listeners are notified and acknowledge the slot was updated
- The listeners also notify the slots they produce of the event, and those slots become 'stale'
- The producers perform their job, return the results and notify the slot of their completion
- The triggering slot compare the listed of producers which acknowledged the change and the producers which returned the results
- If all producers have performed the task, the slot's status becomes 'current'

- Now one of the slots affering to this slot becomes 'stale'. This slot becomes stale too and notified to all producers

2.2 Producer

A producer is a process which, given a list of optional input slots, produces a number of output slots. A producer will always have at least one slot as output. A producer also must define a list of:

1. **Required input slots** These are the minimum data required by the producer to perform the task. A required slot can be **inclusive** or **exclusive**, which means that the slot must not be available to trigger to producer. A required slot can then be **fault_tolerant**, i.e. failed slots are still considered for joining operation, otherwise a failed input slot make stale all produced slots. This differentiation is needed to handle proxies.
2. **Filters** Filters are applied to the slots' key to restrict the Producer activities to a narrower set of slots.
3. **Provided slots** The outputs that the producer commits to provided. The producer assigns to each output slot a priority. No other producer may not output the same slot with equal priority.

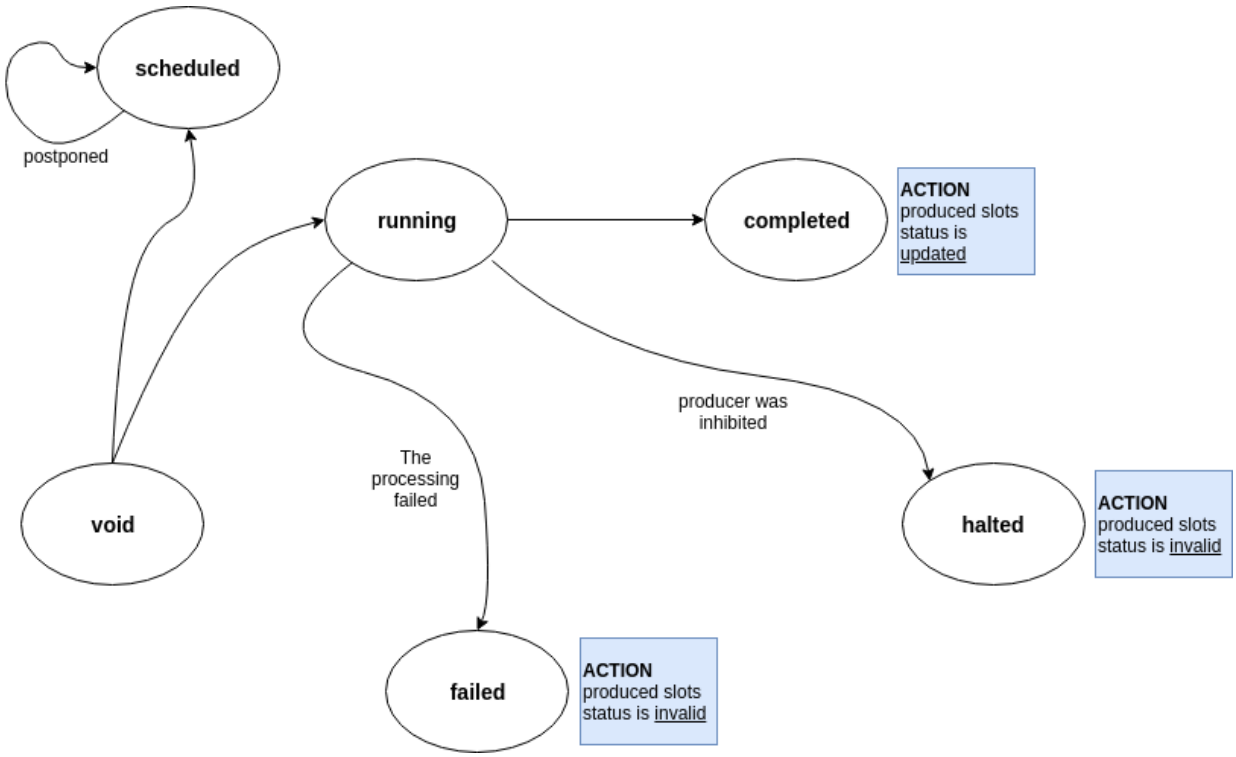
Furthermore, each running producer is identified by a unique id or **pid**

Note: The "Optional Slots" have been **removed**. That behavior can be simulating adding new producers having those optional slots as required.

2.2.1 Producer states

The diagram below shows the state diagram for the producer:

Note: The **scheduled** state was introduced to handle methodologies based on aggregation of data (for example proxy). For those, it is pointless kicking-off new producers as soon as new data is available. Better to schedule, i.e. delay, the processing for a while as very likely new data will become available in the meanwhile.



2.2.2 Brainstorming

The producer is a template for actual running methodologies. It establishes relationship with data and how slots are aggregated. But running producers are *instance* of the producer template.

2.2.3 Producer types

There are different type of producers:

- **Demoniac producers** This producer is constantly running in the background. Example of this type of producers is the *Totem Producer*, which continuously polls for new data from Totem
- **Non-demoniac producers** They have limited life span and terminate once all output slots are produced

2.2.4 Producer output

Apart from the output slots, the data generated by the processing, the producer must return the input slots' used for the production of each output slot. This is needed to build a dependency tree. Indeed, once the producer output is processed and output slots' stored in the MRP table, the dependency tree SLOT_TREE must be updated accordingly.

2.2.5 Producer Proxy

All producers are executed through a common interface called *Producer Proxy*. Proxies are divided into

- **Multi-threaded (MT) Proxies** Producers are not executed in a different process. For example, producers which simply aggregates or filters data can be run in a separated thread. Note that producers accessing analytic funcionality (CAL, Zuul) may not be multi-treaded producers

- **Multi-processing (MP) Proxies** Producers are executed in their own process and potentially in a different machines

The Proxy is responsible for join together all the slots needed for the producer to run. A set on joined slots constitute an *parcel*. An parcel is a collection of slots required to produce at least one output slot. For example, implied volatility and forward for asset A constitutes a parcel for the volatiity fitting producer

A further differentiation is in

- ***Independent* Proxies** Producers are executed as soon as parcels are available, but subject to batch sizes MIN_BATCH.

Todo: Add example of Independent Proxies

- ***Dependent* Proxies** Producers are triggered once all the input slots' producers are terminated. This type of proxy is needed to execute methodologies based on the whole aggregated data, for example calculation of VarSwap basis. For Dependent Proxies MIN_BATCH=9999

In order for Proxy to work, they must:

- Receive notification of changes in status of required and optional slots
- Aggregate the slots into parcels.
- Trigger the actual producer once a set of conditions are met

The conditions to be satisfied to trigger a calculation are:

- The number of available parcels are more than MIN_BATCH > 0.
- The number of available parcels is less than MIN_BATCH but the parcels have been available for more than MAX_WAIT_TIME.
- The proxy is Dependent and all the input slots' producers have terminated

2.2.6 A journey inside the producer life!

A slot is completed. It notifies this to the producer, which is one of the listeners, by communicating the sid. The producer acknowledges it (do we really need this? **Yes I do!** This is required to sort out the issue with notifying producer with updated data) and registers the new available data. It then checks whether it has received the required number of slots. If it does, then parcel is sent for processing. Now, it can happen that one of the input slots becomes invalid, or than a new optional slot becomes available. In this case the producer is inhibited. A signal is sent to the running producer. Once the producer returns the output, either because it stopped processing upon inhibition signal or because it terminated is job, that output is simply neglected but the output slot must be set to **invalid.** Why we need this? Not sure, let's assume we need to flag the failure.

2.2.7 Joining slots into parcels - proposal

1. a slot is received and stored in the parcels_table
2. consider all the producers affected
3. join the required slots together
4. **projects** the keys on the producer aggregation keys. This is the key point: we know upfront which producers are affected
5. select all the slots with keys on each distinct projection

6. If a producer with that aggregation still running then inhibits it before starting the new one

---

Note: steps 2-6 must be in a separate function, which may be possible to call for example by a scheduler.

Note: to joining at step 3 may not be required, but at the moment it seems required to process batches of different slots, for example for proxies calculation

---

Warning: A tricky aspect is that the calculation is triggered by an event! So for example this makes difficult to restart from a crash. This can be addressed leveraging on the slot **updated** status. So slots which are **pending** can have their status overridden to **updated**.

2.2.8 Examples

VarSwapVolBasis calculation

```
required_slots = (Totem.VarSwapVolBasis,)
output_slots = (Totem.VarSwapVolBasisProxy,)
producer_key = 'product_type|region'
```

CHAPTER

THREE

MRP DB

The MRP database is needed in order to:

- Synchronize the access to the data from different processes running of different machines
- Persist the slot's states
- Keep track of slot's dependency trees
- Keep track of running producers
- Joining the slot's data to be feed to the producers

In the following, the required table are listed

3.1 underlyings

- uid
- uname
- ric
- currency
- class
- type
- region

3.2 slot_classes

Information of the slot, with columns:

- **SLOT_NAME** Name of the slot, for example MarketData.Vanilla.Totem
- **SLOT_ID** Identifier
- **KEY_MAP** This is a string made up of the slot's keys joined by '/, for example 'uname/region/type'

3.3 producer_classes

Information on the producer.

3.4 SLOT_DEPS

Information on slots' dependencies:

- **SLOT_ID** the id of the slot
- **DEP_ON_ID** The slot SLOT_ID depends on
- **REQUIRED** The dependency is REQUIRED (1) or OPTIONAL (0)

3.5 SLOT_TABLE

Keeps track of existing slots and register them with the SID. Columns are:

- **SLOT_ID** id of the slot, a foreign key into SLOT_META
- **DATE** The valuation date
- **SLOT_KEY** This is a string made up of the slot's keys joined by '/, for example '0123456/EURO/INDEX'. The semantic is provided by they SLOT_META.KEY_MAP
- **SID** unique identifier
- **SSTATUS** slot's status
- **FILE** path to the file containing the data

3.6 SLOT_TREE

Slot's dependency tree. This is required to invalidate downstream slots (state set to 'stale') upon slot's update.

- **SID1** SID of the slot
- **SID2** SID of a slot depending on SID1

Note: This table is not a duplicate of SLOT_DEPS. The latter describes they dependencies between slots through methodologies, the former the actual dependency on the produced slots

---

3.7 RUNNING_PROD

The table of the running producers

- **PROD_NAME** The producer name
- **PID** The producer unique id
- **START_TIME** The time when the producer was kicked off
- **SID** The slots being produced

3.8 SLOTs Tables

Each slot is associated with a table with same name as the slot, with '.' replaced by '_'. The tables have columns which are specific to the relative slot, apart a few which are fixed:

- **DATE** The valuation date
- **SID** The id from the SLOTS_TABLE
- **LAST_UPDATED** The last update time
- **PRODUCER** The producer that created the data
- **PRIORITY** The producer's priority
- Set of specific keys that identifies the item itself. In SQL parlance, these are *primary keys*.

PRODUCERs Tables Each producers is associated with a list of slots which where notified to the producers and accepted.

- **SID** the id from the slot_table
- **PSTATUS** status of the processing ('pending','processing')

3.9 parcels_table

Contains the Information that the producer proxy need to bundle the slots into parcels and kick-off tasks.

So far we have:

- **pid** The producer id
- **sid** The slot id acknowledged by the producer
- **uname, currency, class, type, region** key from the slot, some could be None

CHAPTER

FOUR

THE SLOT PSEUDO CODE DESCRIPTION

4.1 The monitor pseudo code

The monitor is that piece of code that monitor the *buffer* folder. In this folder, producers drops items and the monitor processes that items and plug them into the slots. :: Upon arrival of new items, the slot table will be updated setting the new status as either 'updated' or 'failed'. It is not in the mandate of the Monitor to perform further processing, which is left for the SlotTable.flush() method.

```
# MAX_PROCESSED_ITEMS is the maximum number of incoming items that
# can be processed by this machine's instance
MAX_PROCESSED_ITEMS = 10
# This piece of code monitor for incomining files in the
# buffer folder
def process():
        processed_items=0
        with buffer_lock(): # wait until it gains access to the directory
                with mrp_lock():
                        for f in glob.glob('*.json'):
                                id = build_id_from(f)
                                fname = build_file_name(f)
                                if f.status == 'ok:
                                        SlotTable.updated(id, fname)
                                else:
                                        SlotTable.failed(id)
                                move_file_from_to(f, fname)
                                processed_items += 1
                                if processed_items == MAX_PROCESSED_ITEMS:
                                        break
```

4.2 The slots pseudo code

```
# This piece of code manages the update of the slots status
# It is responsible to
# 1. create new slots
# 2. set the status to 'pending' or to 'invalid'
# 3. It should also the the status to 'pending' and 'current'

# The status 'updated' is necessary as it is not obvious that there are
# spare producers able to accept the work.
# The 'failed' state is needed too - only once all dependent slots have
# been invalidated the state can go to 'invalid'
```

```
SSTATUS = ['updated', 'current', 'stale', 'pending', 'failed', 'invalid']
class SlotTable():
        """
        This is the SLOT_TABLE.
        It has 3 public member functions:
        set_to_updated(s,fname) : set the status to 'updated'
        set_to_failed(s) : set the status to 'failed'
        flush() : process all the changes
        get_slot_id(s) : produce a unique slot it
        """
        sid = None
        key = None # concatenation of keys
        slotname = None
        file = None # path to the file containing the data
        sstatus = None # this is the slot status in SSTATUS
        updatetime = None # the time stamp of last update

        def _slot_to_id(self,s):
                '''
                concatenate all the slot s attributes, joining by '.'
                '''
                res = s.name + "/" + str(s.date)
                values = []
                for k in s.keys():
                        values.append(s.k)
                return res + '/' + sorted(valued).join('/')

        def _slot_exist(self,s):
                '''
                Return True is s is in the table, False otherwise
                '''
                id = _slot_to_id(s)
                sid = select('sid').where(sid == id)
                if len(sid) == 1:
                        return True
                return False

        def _add_slot(self, s):
                '''
                Insert the slot s in the table with status='void' as return the sid
                '''
                if self._slot_exist(s):
                        raise Exception("Slot already exist")
                id = create_id()
                SlotTable.update(date=s.data, slotname=s.name, sstatus='void', sid=id)
                return id

        def get_slot_id(self,s):
                '''
                concatenate all the slot s attributes, joining by '.'
                check if the sid is already available
                if it is, return it otherwise create a new one,
                insert into the table and return it
                TODO: how do we treat date? Should be a part of the key
                '''
                if self._slot_exist(s):
                        return SlotTable.select('sid').where(sid == id).fetch_one()
                raise Exception("Slot not existing")
```

```
        def _update(self, s):
                '''
                The item was updated.
                All listeners are notified and then the item's status is set to
↪'pending'.
                The status of dependent slots are set to 'stale'
                NB: The slot id always exists as it is created once the producer␣
↪accepts the incoming slots
                and update the SlotTree table
                '''
                sid = get_slot_id(s)
                listeners = get_listeners_of(s.name)
                for l in listeners:
                        l.accept(s)
                set_to_pending(s)

        def _invalidate_dependents(self, sid):
                invalid_sids = SlotTree.select('sid2').where(SID1=sid)
                for sid in invalid_sids:
                        set_to_stale(sid)
                        for l in get_listener(sid.name):
                                l.remove(sid)
                        _invalidate_dependents(sid)

        def _failed(self,s)
                '''
                The item processing failed.
                The status of dependent slots are set to 'stale'
                '''
                self.set_to_failed(s)
                s.file = None
                sid = self._get_slot_id(s)
                _invalidate_dependents(sid)

        def set_to_updated(self, s, fname):
                sid = self._get_slot_id(s)
                self.where(sid==sid).update((status,file)=('updated',fname))

        def set_to_failed(self, s):
                sid = self._get_slot_id(s)
                self.where(sid==sid).update((status,file)=('updated',fname))

        def flush(self):
                for u in self.select(sid).where(status=='updated'):
                        self._update(u)
                for f in self.select(sid).where(status=='updated'):
                        self._failed(f)
```

4.3 The ProxyProducer pseudo code

```
class ProxyProducer(object):
        def __init__(self, producer):
                self.producer_name = producer
                self.required_slot_num = len(producer.required_slots)
```

```
                self.optional_slot_num = len(producer.optional_slots)
                self.keys = producers.keys

        def accepted(self, sid):
                """
                Return True if the sid is already registered in the table
                """

        def accept(self, slot):
                """
                Take the incoming slot, join it with the producer keys and insert it␣
↪in the
                producer table
                """

        def launch(self):
                """
                1. take all distinct combination of producers keys
                2. for each combination compute the number n of required slots
                3. if n>self.required_slot_num and at east one row is not pending then
                        1. set all row to processing
                        2. kick-off the producer task
```

4.4 The SlotTree pseudo code

```
class SlotTree():
        SD1 = None
        SD2 = None
        def add(self, sd1, sd2):
                '''
                add the entry (sd1,sd2) to the table
                '''
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search